MASTER IN COMPUTER SCIENCE

Schema modification propagation for relational database applications

Wagner, Sandra; Schmit, Luc

Award date:
1995

Awarding institution:
University of Namur

Facultés Universitaires Notre-Dame de la Paix
Institut d'Informatique
rue Grandgagnage, 21
5000 Namur

SCHEMA MODIFICATION PROPAGATION FOR RELATIONAL DATABASE APPLICATIONS

Wagner Sandra & Schmit Luc

Supervisor: J-L. Hainaut

Thesis presented in order to obtain the
degree of master in computer science

Academic Year 1994-1995

ABSTRACT

Database evolution is the ability of a database system to respond to changes in the real world by allowing its schemas to evolve. Our thesis will use the ER (Entity-Relationship) and the relational models for representing the data structures at the different levels and it will give a typology of the modifications offered on the conceptual level. In order to analyse these modifications, the thesis will follow a systematic and inductive approach. Indeed, for each modification, a detailed study of its impact on the logical level, the SQL description, the data and the application programs will be presented. Moreover, these modifications will first be applied on a case study and will only then be treated in general. Furthermore, some indications are given on how to integrate the studied modifications into a CASE tool offering database evolution facilities.

L'évolution des bases de données est la capacité d'un système de bases de données de répondre aux changements du monde réel en permettant l'évolution de ses schemas. Notre mémoire utilisera le modèle EA (Entité-Association) et le modèle relationnel pour représenter les structures des données aux différents niveaux et donnera une typologie des modifications offertes au niveau conceptuel. Afin d'analyser ces modifications, le mémoire adoptera une approche systématique et inductive. En effet, pour chacune des modifications, une étude détaillée analysera les impacts au niveau logique, sur la description SQL, les données et les programmes d'applications. De plus, ces modifications seront d'abord étudiées sur une étude de cas avant d'être traitées en toute généralité. En outre, le mémoire contient quelques indications quant à l'intégration des modifications étudiées dans un outil CASE offrant des fonctionnalités supportant l'évolution des bases de données.

ACKNOWLEDGEMENTS

It is a great pleasure for us to acknowledge the assistance and contributions of a large number of individuals. First, our supervisor J-L. Hainaut who continuously encouraged and motivated us to complete our thesis. He constantly reviewed portions of our thesis and suggested various improvements.

We would also like to thank our supervisor J.F. Roddick from the 'University of South-Australia' who constantly helped us during our practical training.

We would also like to underline our thanks to the DB-MAIN group (V. Englebert, J. Henrard, J-M. Hick and D. Roland) who gave us hints where to find the necessary bibliographies and who continuously supplied us with new versions of the DB-MAIN software.

Finally we gratefully acknowledge the support, encouragement and patience of our families and friends, especially M. Nosbusch.

TABLE OF CONTENTS

Chapter 1 : Introduction

Chapter 2 : Framework of our Thesis

Chapter 3 : Study of the Kernel

Chapter 4 : Study of the Modifications: Case Study Approach

Chapter 5 : Study of the Modifications: General Approach

Chapter 6 : Introduction to the Modifications on the Basic ER Model

Chapter 7 : Introduction to the Modifications on the Rich ER Model

Chapter 8 : Integration into a CASE Tool

Chapter 9 : Conclusion

Appendix 1 : Study of the Modifications: Case Study Approach

Appendix 2 : Study of the Modifications: General Approach

TABLE OF FIGURES

Chapter 1 : Introduction

Chapter 2 : Framework of our Thesis

Chapter 3 : Study of the Kernel

Chapter 4 : Study of the Modifications: Case Study Approach

Chapter 5 : Study of the Modifications: General Approach

Chapter 6 : Introduction to the Modifications on the Basic ER Model

Chapter 7 : Introduction to the Modifications on the Rich ER Model

Chapter 8 : Integration into a CASE Tool

Chapter 9 : Conclusion

Chapter 1:

Introduction

As schema modification is part of the life cycle of a database, we will first describe the life cycle of a database and of its application programs. We will mainly consider the design and maintenance phases as they seem important for our thesis.

In a second section, we will then discuss what has already been achieved in the domain of database evolution. We will distinguish three different areas: database evolution for standard data structures, database evolution for object oriented approaches and historical databases.

1. LIFE CYCLE OF A DATABASE AND OF ITS APPLICATION PROGRAMS

In a large organization, the database system is typically part of a much larger information system that is used to manage the information resources of the organization. An information system includes all resources within the organization that are involved in the collection, management, use, and dissemination of information. In a computerized environment, these resources include the data itself, the DBMS software, the computer system hardware and storage media, the personnel who use and manage the data (database administrator, users, and so on), the application software that accesses and updates the data, and the application programmers who develop these applications. Hence, the database system is only part of a much larger organizational information system.[ELM94, page 450] Every information system has a life cycle and as the database system is part of such a system, it has its own life cycle too.

In this section we will begin with a description of the life cycle of a database system. In further subsections, we will analyse more in detail two of the phases of such a database life cycle: the design and the maintenance phases. The design subsection is essentially based upon [ELM94], whereas the maintenance subsection is largely inspired by [HAI94a].

1.1. DATABASE SYSTEM LIFE CYCLE

The life cycle of a database system is represented in Figure 1-1.

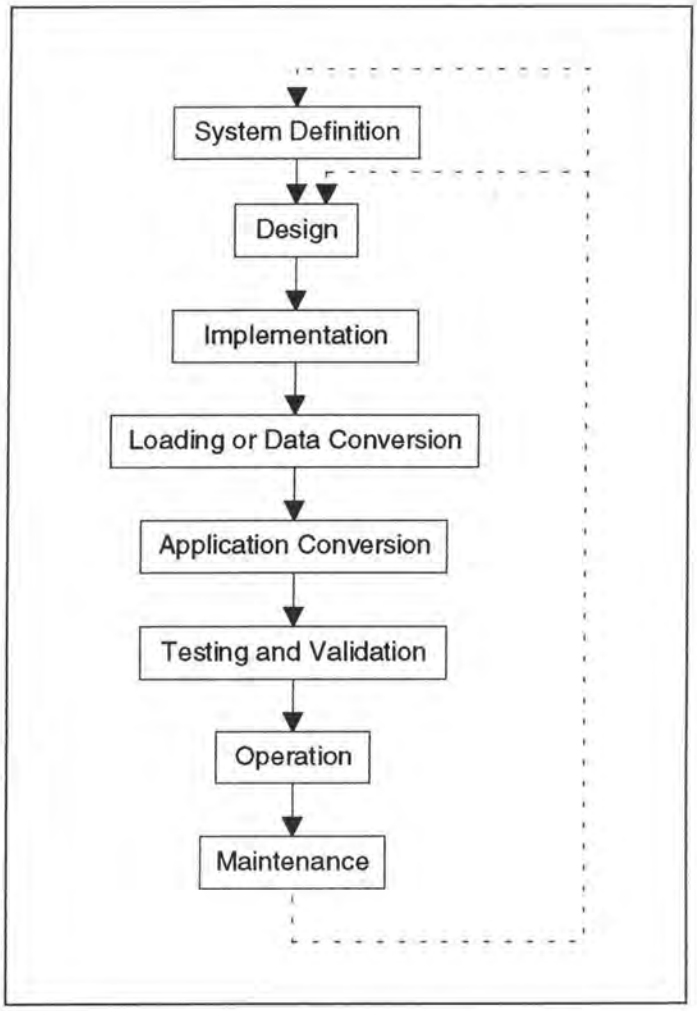


Figure 1 - 1 : Database system life cycle

We will now briefly explain each of the phases of the life cycle of a database system.

- *System Definition:*
  The scope of the database system, its users and its applications are defined.

- *Design:*
  At the end of this phase, a complete conceptual, logical and physical design of the database on the chosen DBMS is ready.

- *Implementation:*
  This comprises the process of writing the corresponding database definitions, creating empty database files and implementing the software applications.

- *Loading or Data Conversion:*
  The database is populated either by loading the data directly or by converting existing files into the database system format.

- *Application Conversion:*
  Any software applications from a previous system are converted to the new system.

Note that the conversion steps are not applicable when both the database and the applications are new. When an organization moves from an old established system to a new one, these tend to be the most timeconsuming and the effort to accomplish them is often underestimated.

- *Testing and Validation:*
  The new system is tested and validated.

- *Operation:*
  The database system and its applications are put into operation.

- *Maintenance:*
  During the operational phase, the system is constantly maintained. Growth and expansion can occur in both data content and software applications. Major modifications and reorganisations may be needed from time to time. Those modifications involve returns to the 'System Definition' and 'Design' phases, illustrated by the dotted arrows in Figure 1-1 (see page 1-2).

In the next subsections, we will analyse more in detail the 'Design' and 'Maintenance' phases as they are important for the further developments of our thesis.

1.2. THE DATABASE DESIGN PHASE

We now focus on the second step of the database system life cycle, which we call database design. The problem of database design can be stated as follows: Design the conceptual, logical and physical structure of one or more databases to accomodate the information needs of the users in an organization for a defined set of applications.

The goals of database design are multiple: to satisfy the information content requirements of the specified users and applications; to provide a natural and easy-to-understand structuring of the information; and to support processing requirements and any performance objectives such as response time, processing time, and storage space. These goals are very hard to accomplish and measure. The problem is aggravated because the database design process often begins with very informal and poorly defined requirements. By contrast, the result of the design activity is a rigidly defined database schema that cannot easily be modified once the database is implemented. We can identify six main phases of the database design process:

1. Requirements collection and analysis
2. Conceptual database design
3. Choice of a DBMS
4. Logical database design
5. Physical database design
6. Database system implementation

The design process consists of two parallel activities, as illustrated by the two last columns in Figure 1-2. The first activity involves the design of the data content and structure of the database; the second relates to the design of database processing and software applications. These two activities are closely intertwined. For example, we can identify data items that will be stored in the database by analysing database applications. In addition, the physical database design phase, during which we choose the storage structures and access paths of database files, depends on the applications that will use these files. On the other hand, we usually specify the design of database applications by referring to the database schema constructs, which are specified during the data content and structure design. Clearly, these two activities strongly influence one another.

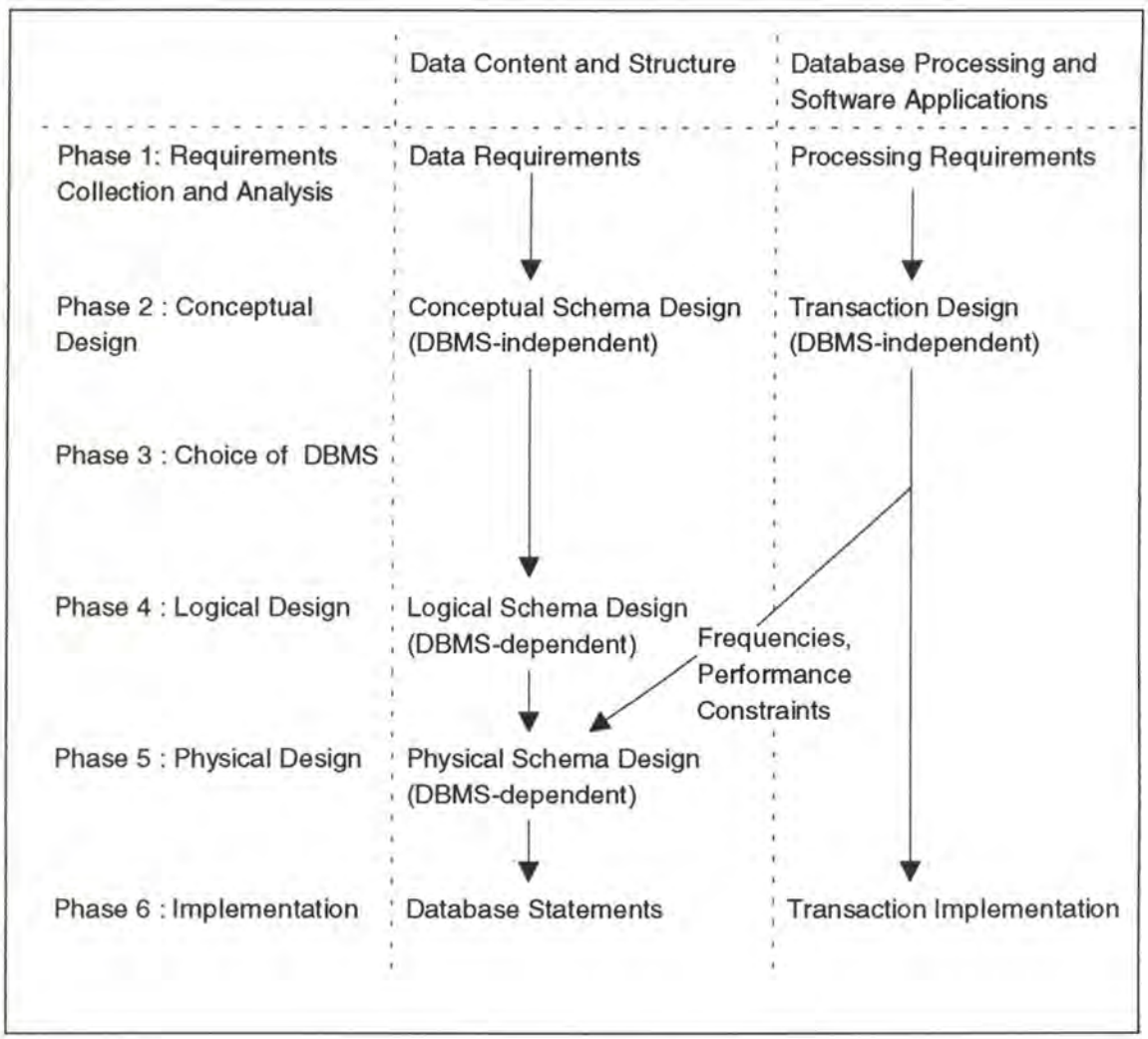


Figure 1 - 2 : Phases of database design

The six phases mentioned above do not have to proceed strictly in sequence. In many cases we may have to modify the design from an earlier phase during a later phase. These feedback loops among phases -and also within phases- are common during database design. We do not show feedback loops in Figure 1-2 to avoid complicating the diagram. Phase 1 in Figure 1-2 involves collection of information about the intended use of the database, whereas phase 6 concerns database implementation. Phases 1 and 6 are sometimes considered not to be part of

database design per se, but part of the database system life cycle. The heart of the database design process is composed by phases 2, 4 and 5, which we briefly summarize here:

- *Conceptual database design (phase 2):*
  The goal of this phase is to produce a conceptual schema for the database that is independent of a specific DBMS. We often use a high-level data model such as the ER (Entity-Relationship), EER (Extended Entity_Relationship) or NIAM (Nijssen Information Analysis Method) model during this phase. In addition, we specify as many of the known database applications or transactions as possible, using a notation that is independent of any specific DBMS.

- *Logical database design (phase 4):*
  During this phase we map the conceptual schema from the high-level data model used in phase 2 into the data model of the DBMS chosen in phase 3. We can start this phase after choosing an implementation data model, rather than waiting for a specific DBMS to be chosen -for example, if we decide to use some relational DBMS but we have not yet decided on a particular one. We call the latter system-independent (but data model-dependent) logical design.

- *Physical database design (phase 5):*
  Physical database design is the process of choosing specific storage structures and access paths for the database files to achieve good performance for the various database applications. Each DBMS offers a variety of options for file organization and access paths. These usually include various types of indexing, clustering of related records on disk blocks, linking related records via pointers, and various types of hashing. Once a specific DBMS is chosen, the physical database design process is restricted to choosing the most appropriate structures for the database files among the options offered by that DBMS. The following criteria are often used to guide the choice of physical database design options:

  1. *Response Time:*
     This is the elapsed time between submitting a database transaction for execution and receiving a response.

  2. *Space Utilization:*
     This is the amount of storage space used by the database files and their access path structures.

  3. *Transaction Throughput:*
     This is the average number of transactions that can be processed per minute by the database system.

1.3. THE DATABASE MAINTENANCE PHASE

Most database systems at some time or another require a change to their schema, due to either changes in the real world, a change in the application requirements or mistakes during system analysis or design [ROD94, page 1], regrouped by the term of 'changes in the requirements'. In order to study the impact of the changes in the requirements we will consider the following abstract framework.

Let us consider a database that satisfies requirements R0. This database comprises schema S0 and, at a given instant, data D0. Schema S0 is made of the physical schema PS0, the logical schema LS0 and the conceptual schema CS0. A set of database applications P0 have been built for the database; they all work on the data through schema PS0.

Let us consider that requirements R0 have changed into R1. In most cases, this change is translated into modifications of schema S0 (CS0, LS0 and/or PS0) leading to the new schema S1 (CS1, LS1, PS1). If one of the schemas (CS0, LS0 or PS0) has been changed, the others must be changed accordingly. Data D0 is no longer valid, and has to be converted into data D1. Finally, the application programs P0 must be partly rewritten in order to comply with the new data structures described in PS1. This situation is depicted in Figure 1-3.

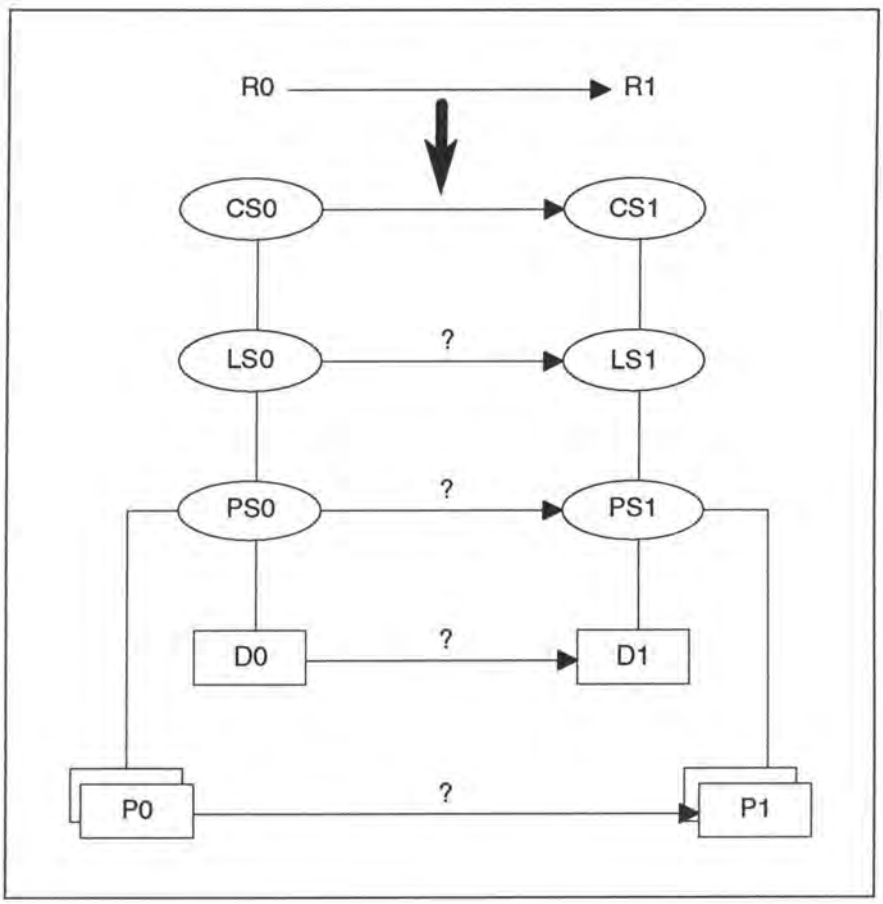


Figure 1 - 3 : Representation of the database evolution problem

Three typical maintenance strategies, supporting the translation of the changes, can be distinguished in order to respond to major practical problems that will occur:

- Forward Database Maintenance
- Backward Database Maintenance
- Anticipating Design

1.3.1. Forward Database Maintenance

The first strategy is situated in a somewhat idealistic context in which the schemas CS0, LS0 and PS0 are available and correct[1]. Let us suppose that the change in requirements R0 has been translated into a change in the conceptual schema CS0 in such a way that CS1 now satisfies R1 (e.g. adding an attribute, changing the cardinality of a role and so on). The basic problem is how this change could be (as far as possible automatically) propagated down to the logical and physical schemas LS0 and PS0, data D0 and application programs P0 to produce LS1, PS1, D1 and P1. This problem can be called the forward database maintenance problem, a reference situation against which all the other problems will be analysed.

A schema modification occurs as a consequence of a change in the requirements of (the database component of) the information system. First, the relevant schema must be retrieved. If the requirements describe the user expectations (functional requirements), their change will be converted into a change in the conceptual schema. On the contrary, if they specify technical constraints or goals of the final system (non functional requirements), they will concern lower level schemas, in which criteria such as performance and security are addressed.

Automatic or assisted update propagation requires the knowledge of the exact mapping between the conceptual, logical and physical schemas. Indeed, this mapping allows to replay[2] most processes that were carried out when building the old system. The designer is only responsible for controlling the propagation of the new constructs. When the new schema has been produced, it should be as close as possible to the old one, except for the structures that have been concerned by the changes. Following this schema updating, the database contents must be converted as well. According to the kind of change, some data will be discarded, some will be converted, while some data structures will be left empty. Finally, the application programs will be updated in order to let them operate on the new database.

1.3.2. Backward Database Maintenance

Let us suppose that the conceptual and logical schemas CS0 and LS0 are still available and correct[3]. Very frequently, due to time constraints and current CASE tools weaknesses, the change in requirements R0 are directly translated into changes on the physical schema PS0 in such a way that PS1 now satisfies R1 (e.g. adding a column to a table). The problem here is how these changes in PS0 could be propagated up to the conceptual and logical schemas CS0 and LS0, giving CS1 and LS1 that reflect the semantics of the new physical schema PS1.

---

[1] In many situations, particularly when so-called legacy-systems are considered, the only description available is the source code of the file and database description and of the manipulation programs. Schema CS0, and sometimes (for standard file applications for instance) schema LS0 and PS0 are missing. In order to make the system evolve securely, we have first to recover schemas PS0, LS0 and CS0 before applying the forward database maintenance strategy. This way of proceeding is called database reverse engineering.

[2] The term of replay is used in software engineering to designate the reexecution of design activities that have been carried out to produce the former version of a system.

[3] In many situations, particularly when so-called legacy-systems are considered, the only description available is the source code of the file and database description and of the manipulation programs. Schema CS0, and sometimes (for standard file applications for instance) schema LS0 and PS0 are missing. In order to make the system evolve securely, we have first to recover schemas PS0, LS0 and CS0 before applying the backward database maintenance strategy. This way of proceeding is called database reverse engineering.

The backward database maintenance problem consists thus in translating the evolution of the database into changes in the physical file and database structures, then in propagating these changes up to the logical and conceptual schemas. Backward maintenance is certainly not the cleanest way to support database evolution. However it corresponds to common practice.

Though it seems linked to database reverse engineering, this problem is significantly different in two aspects. On the one hand, the conceptual and logical schemas of the old version of the database are available and correct. On the other hand, most variations between old and new physical schema versions are minor: modify a column domain, add, remove or rename a column and so on.

Knowing how each conceptual and logical construct has been translated into physical database structures (forward mapping), one can deduce the backward mapping, i.e. from what conceptual and logical constructs a given physical construct was derived. Modifying the physical schema must trigger a modification of the old conceptual and logical schemas in such a way that the new conceptual and logical schemas would lead to the new physical schema, should the same translation rules as earlier be applied. The data conversion, as well as program conversion, can be tackled through forward maintenance. Note however that these conversions have often been immediately carried out .

1.3.3. Anticipating Design

It is widely accepted that some design and programming styles lead to better tolerance to requirements evolution than others. Hence the following problem: what reasonings and design techniques can be proposed to designers and programmers to build database structures (CS0, LS0 and PS0) and applications (P0) that are more robust against changes of requirements? This problem relates to database design methodologies in which the stability of the programs w.r.t. changes in data structures is considered as a high priority criterion.

1.4. CONTEXT OF OUR THESIS

The topic of our thesis is 'Schema Modification in Relational Database Systems' and can be situated in the maintenance phase of a database life cycle. Our thesis will be developed in the idealistic context of the forward database maintenance strategy (see page 1-7) in which the conceptual, logical and physical schemas are available and correct. We will thus only study modifications on the conceptual schema and their impact. Before delimiting the framework of our thesis in the second chapter, we will briefly analyse the state of the art of the database maintenance domain.

2. STATE OF THE ART

As we already said in the previous part, most database systems at some time or another require a change to their schema, due to either changes in the real world, a change in the application requirements or mistakes during systems analysis or design. When these changes occur database systems must provide schema manipulation tools with which the database administrator can modify the database. In many systems available commercially however, the database administrator must also make decisions on whether the data already held in the database is valid given the new schema. In many cases, data is either deleted unnecessarily, misleadingly left in the database or the schema is made unnecessarily complicated by the retention of obsolete attributes.[ROD92, page 1]

To avoid these situations, research has started, over the last few years, to investigate how to facilitate database maintenance. In order to keep an overview of the work that has been done so far, we categorise it broadly into three areas:

- Database Evolution for Standard Data Structures
- Database Evolution for Object Oriented Approaches
- Historical Databases

Note that we do not pretend to be exhaustive, as database evolution has raised into a vast domain during the last few years.

Before discussing these three areas in detail, we have to specify first a certain number of concepts. [ROD93] tried to define schema modification, schema evolution and schema versioning.

- *Schema Modification:*
  Schema modification is accommodated when a database system allows changes to the schema definition of a populated database.
- *Schema Evolution:*
  Schema evolution is accommodated when a database system permits the modification of the database schema without loss of the semantic content of existing data.
- *Schema Versioning:*
  Schema versioning is accommodated when a database system allows the viewing of all data, both retrospectively and prospectively, through user definable version interfaces.

Note that if in future we will speak about database evolution, then we will not reference any of the three terms specifically. Let us now discuss schema and data conversion mechanisms which are used in schema versioning.

A number of suggestions have been proposed for the conversion of the schema at the physical level. Firstly, the complete schema can be converted to a new version. This method, while being conceptually simple, prohibits the parallel schema versions required in some application environments. Secondly, database evolution is achieved through view creation. This second approach allows multiple concurrent versions of the schema.

The mechanisms whereby the data is physically converted to the new version have also been investigated. The three options proposed are the strict conversion method in which a change to the schema results in an immediate propagation of that change to the data, the lazy conversion mechanism in which data are changed to the current format only when required, and the logical conversion method in which the attribute is translated into the required format at access time. No conversion is therefore required.[ROD93, page 3]

After having introduced some concepts, let us study in detail the three previously mentioned areas which categorise the work that has been achieved so far.

2.1. DATABASE EVOLUTION FOR STANDARD DATA STRUCTURES

In the area of database evolution for standard data structures, a lot of research has been done so far, especially for the ER and NIAM high-level models. In [ROD93] we can find a taxonomy for schema versioning based on the relational and ER models whereas in [EWA93] we can find a procedural approach to schema evolution in the NIAM model.

[ROD93] proposes two taxonomies of modifications: one for the relational model and one for the ER model. For each modification in the ER taxonomy the corresponding relational changes are given. For example, the modification *add an entity* involves the operation *create a relation* on the relational level.
[ROD93] has established the relational taxonomy in such a way that the schema modifications are as symmetric and reversible as possible. A schema modification is said to be symmetric if data D0, recorded under the schema S0, can be viewed through the new schema definition S1 and if data D1, recorded under schema S1, can be viewed under the previous schema S0. This is illustrated by the dotted arrows in Figure 1-4.

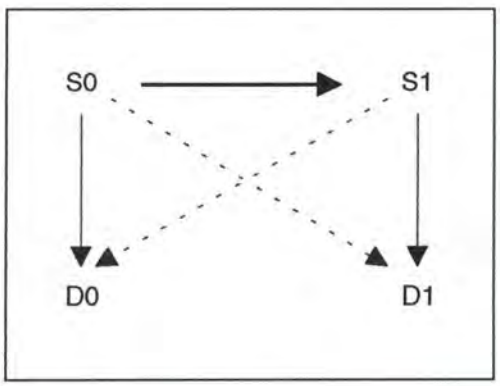


Figure 1 - 4 : Symmetric schema modification

Schema modifications should be reversible in order to allow erroneous changes to be removed. This reversibility is for instance obtained by proposing *deactivate modifications* instead of *remove modifications*. For example, a relation that is deactivated (and which is thus not physically removed) can be recovered through the modification *activate a relation*.

[EWA93] however tackles the database evolution problem in the NIAM model. In this model, failure to identify and remove derivable fact types could lead to unnormalized relations due to

hidden transitive dependencies. The conceptual schemas that will become incorrect because of such failures will produce incorrect relational schemas. [EWA93] therefore proposes for example procedures for safe adding and removing of fact types. Indeed, they check that, in case of an addition, the new fact type does not conflict with existing constraints and does not involve fact type redundancy. They therefore use , among other techniques, functional dependency diagrams. Finally, note that besides ad hoc modifications, such as adding or removing a fact type, [EWA93] considers schema integration as an evolution process too. During the schema integration process, an integrated schema is constructed starting from one or more source schemas. This integration is done by choosing one schema as the target schema, and adding the others to it through a series of ad hoc modifications.

2.2. DATABASE EVOLUTION FOR OBJECT ORIENTED APPROACHES

Before treating database evolution in an object oriented environment, it may be good to review the principle concepts of the object oriented data models.

2.2.1. Object Oriented Concepts

The principle concepts of the object oriented data models are: object, class and inheritance.

Object (or Object Instance):
The concept object is akin to that of entity when it covers semantic aspects. An object is defined through the set of attributes that characterise it and the operations it supports. An object has an identity that is independent of its value.

Class:
The concept of class allows the grouping together of objects which have the same data structure (attributes) and the same operations called methods. A class is generally described by a name, its attributes and methods. Classes are organised in a hierarchy of inheritance.

Inheritance:
Inheritance is a mechanism that allows the factorization of the parts common to several classes. Regarding modelling, the inheritance enables one to refine the definitions of classes by introducing a specialization/generalization link. There are several types of inheritance, the best known are: simple and multiple inheritance. The former corresponds to a class hierarchy: each class that is at the ith level of the inheritance tree inherits the attributes and methods of the parent-class at the (i-1)th level (called superclass). In the case of multiple inheritance, classes are arranged in a graph (without cycle): a class can then inherit from several superclasses. However, when the mechanism of multiple inheritance is applied, a conflict of inheritance may arise when the involved superclasses contain attributes or methods that have the same name. Several strategies to solve this kind of conflict can be found in the literature.[BEL93, page 41]

2.2.2. Database Evolution

We will now review some database evolution possibilities provided by some object oriented database systems. We will not realise here a exhaustive study of existing research efforts but only a limited survey of the ability of some OODBMS (GemStone, Orion, Encore, ...) to support the changes on schemas and their propagation on the objects. No system provides a

full support for object evolution; most of them support changes in the class definitions. However, few of them have the ability to propagate these changes to related objects.

Change operations have to ensure that a structurally-consistent schema is produced as a result of the update operation.[BEL93, page 45] For example in GemStone, changing the name of an attribute in a class is propagated to its subclasses, provided they do not redefine it locally. Adding a new attribute is allowed if it is not already defined in a subclass. Deleting an attribute in a class definition is allowed if it is not inherited from a superclass. Further, the deletion is not propagated to the subclasses. This must be explicitly performed on each subclass. In contrast to Orion, a class may not be deleted in GemStone and Encore if there are any existing objects, since no references to deleted classes and objects are allowed. Classes referencing deleted ones are thus forced to refer to their immediate superclass.[NGU89, page 53]

Structural consistency is thus provided by using a set of 'invariants' that define the consistency requirements of the class hierarchy. The main 'invariants' which stemmed from the Orion database system and which are now used in most OODBMS, are:

- *Class lattice invariant:*
  The subclass/superclass relationship forms a lattice having as root a predefined class 'Object'.
- *Distinct name invariant:*
  All attributes or methods defined or inherited must have distinct names.
- *Distinct identity (origin) invariant:*
  All attributes or methods defined or inherited must have a distinct identity.
- *Full inheritance invariant:*
  A class inherits all attributes or methods from each of its superclasses unless it defines an attribute or method with the same name.
- *The type compatibility invariant:*
  If an attribute A2 of a class C is inherited from an attribute A1 of a superclass of C then the type (or domain) of A2 is either the same as that of A1 or a subclass of A1.

These invariants must be preserved by any change on the schema.

An update operation on a schema is qualified as legal if and only if it ensures production of a consistent schema. Update operations on a database schema can be classified in two categories:

- Changes to class definitions including:
  - add an attribute or method to a class
  - delete an existing attribute or method
  - change the name of an attribute or a method
  - change the type of an attribute

- Modifying the graph of classes
  - add a new class to the graph
  - delete an existing class and its links
  - change the name of a class
  - moving a class in the graph

The problem of schema modification cannot be limited to a set of change operations on the class definitions. The system must also provide capability to control update propagation on the object instances.[BEL93, page 46] The spectrum lies between a fully automatic propagation of the changes and a manual one. The first approach is used in GemStone and Orion, while the second is that of Encore. An explicit convert operator has to be invoked by the user in order to modify, in Encore, an object and conform it to a modified class definition. When propagation is automatic, the delay of effective change propagation to the objects has to be defined. Propagation can be immediate or defered. Immediate propagation is adopted in GemStone. It is called conversion. The impact of schema modifications is immediately implemented on the involved objects. Defered propagation is used in Orion. It is called screening. The side-effects are propagated only when the objects are accessed. The first solution emphasizes consistency and information preservation. It also sacrifices performance. [NGU89, page 54] The second solution emphasizes performance and makes the modification effective at the next access to the object.

Most of the OODBMS's provide schema modification facilities. But they seldom support automatic propagation to the objects. Some work has however already been realised. [Bel93] provides, for example, the means to model the schema legal update operations and their propagation to the associated objects. So far there is however no agreement on how to cope with structure update propagation toward the objects.

2.3. HISTORICAL DATABASES

Temporal and historical database systems possess the ability to maintain and manipulate historical data. Since many database systems must not only deal with time-varying data but also with time-varying data structure, support for schema evolution is here also required.[ROD93, page1] The conceptualisation and design of a Temporally Oriented Data Model (TODM) should therefore address both the dynamics of data as well as the dynamics of the definition of data. Such a comprehensive TODM (see Figure 1-5) should thus internalise the handling of schema evolution and support thereby a *Temporally Oriented Data Definition* (TODD) (see the left handside of Figure 1-6). TODD is a view of a database structure that allows it to evolve over time and maintain the correspondence between data definitions and the data they govern. [ARI91, page 451] As we have seen in the previous sections, the database evolution problem has already been studied for a lot of systems, but the classical database restructuring does not necessarily preserve historical information in its contemporary context, and in some cases may imply outright loss of data- both contradict the very premise of historical databases.

A TODM must thus represent the data and the way it progresses with the time. A three-dimensional space can therefore be used, as shown in Figure 1-5, where time, object instances and attributes are the primary dimensions of stored data.

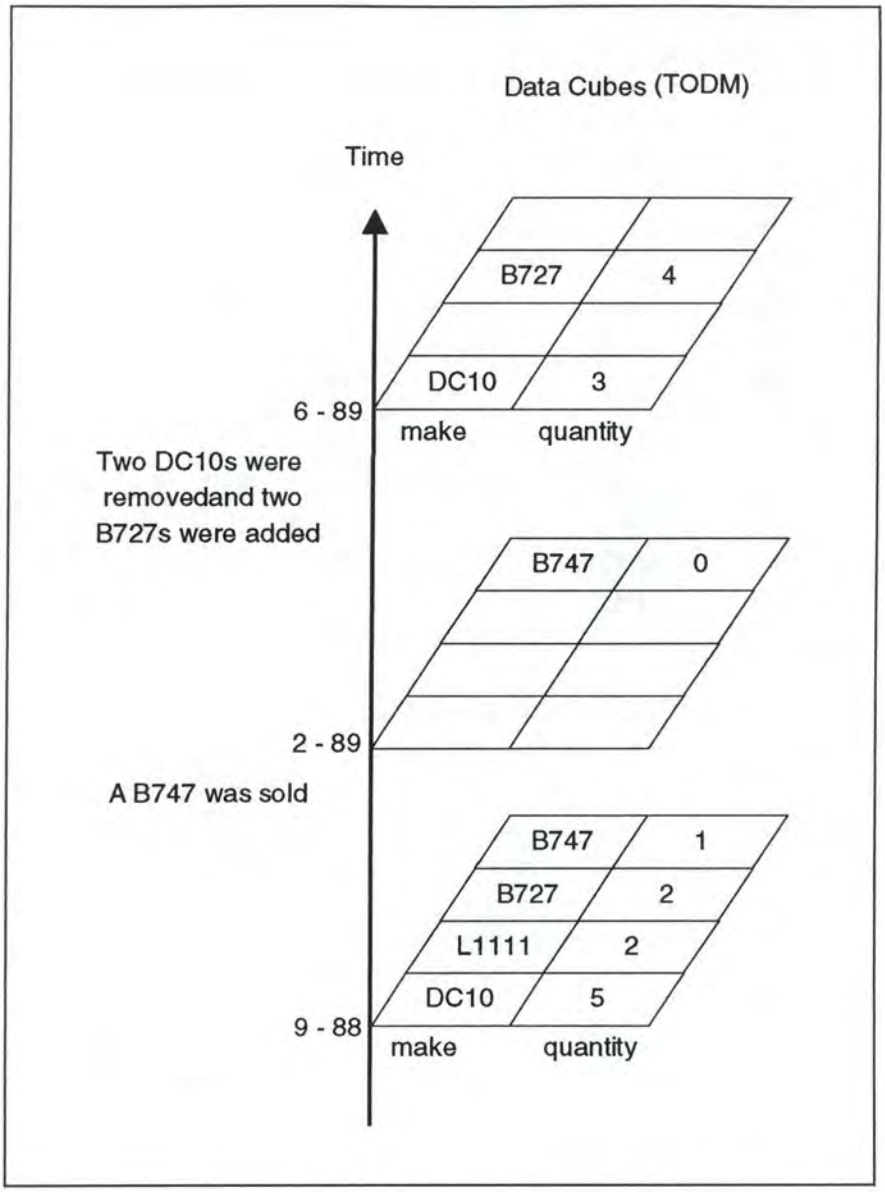


Figure 1 - 5 : A TODM view of an airline's fleet database

A TODM could be extended to capture schema evolution by recognizing two interrelated categories of objects. These categories are:

- *Data objects* (or data cubes) which correspond to entites and relationships meant to be captured by the database, e.g. aircrafts and so on.

- *Schema objects* (or schema cubes) which correspond to data constructs that exist in the database, and which capture their definition.

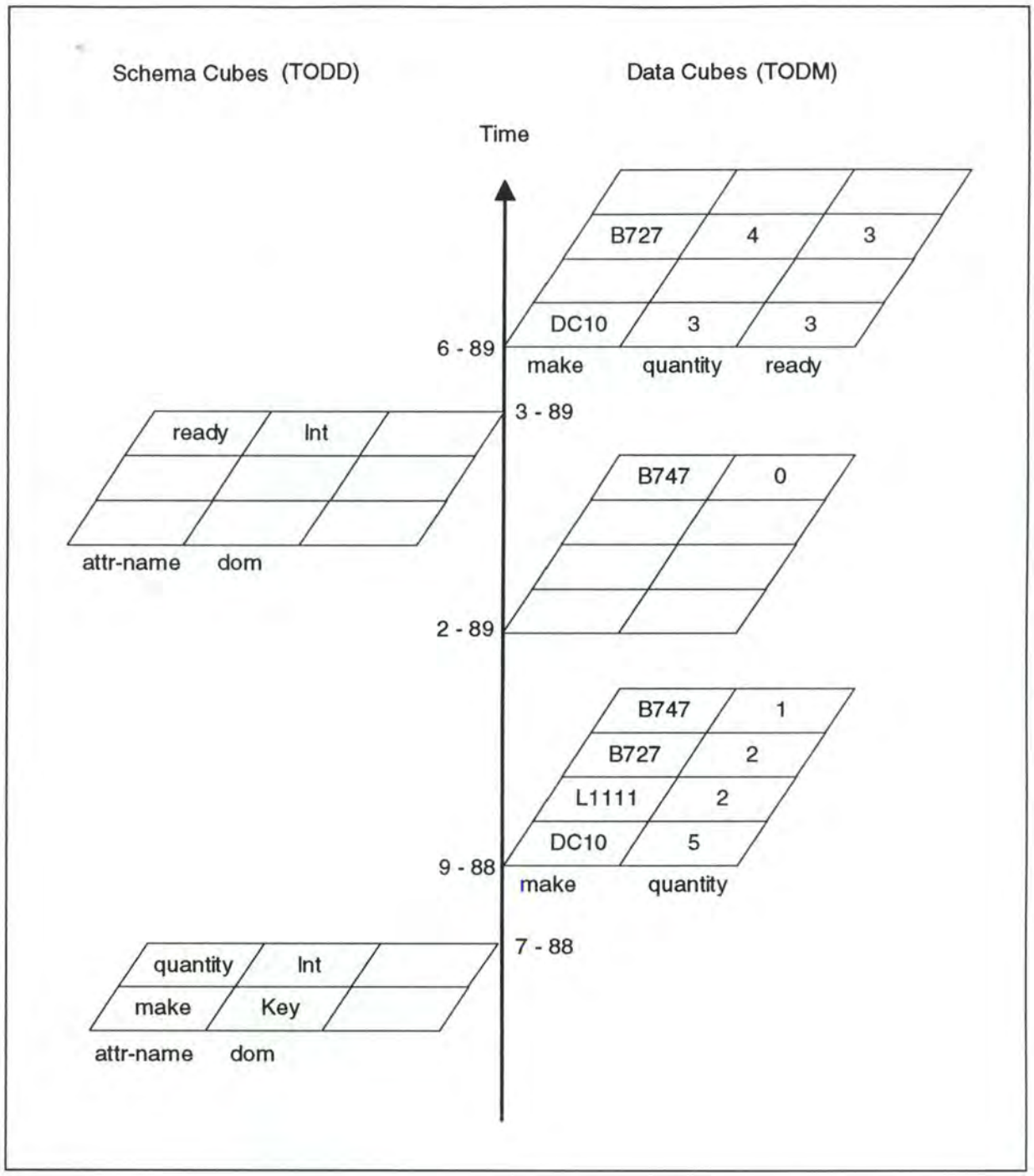


Figure 1 - 6 : A TODM/TODD view of an airline's fleet database

The database has to maintain multiple and time-ordered schemas, through which programs can interpret the portion of data that has been recorded while the corresponding schema prevailed. Operationally, this means that database operations have to refer to the appropriate data definition (schema objects), and that the manipulation of data definitions also needs to be anchored in a temporal context. The conceptual structure of a TODM database with TODD is depicted in Figure 1-6, which augments the view in Figure 1-5 with a TODD.

Chapter 2:

Framework of our Thesis

1. BACKGROUND

As we already said in the first chapter (see page 1-8), our thesis is situated in the maintenance phase of a database life cycle. In order to delimit it we have chosen the forward database maintenance strategy among the three proposed maintenance strategies (see page 1-6). Remember that this strategy uses an idealistic context in which the conceptual, logical and physical schemas are always available and correct and that the forward database maintenance strategy supports changes of functional and non functional requirements. We will however only consider changes of functional ones and we therefore propose, in chapter 3, a typology of modifications on the conceptual level. As we have seen in the state of the art, a lot of different data structure models are possible: we have chosen, for our thesis, the ER model on the conceptual level and the relational one for the levels below. For each modification proposed in the typology, we will study its impact on the logical schema, on the SQL database structure, on the data and on the application programs.

In order to decompose the complex topic of schema modification into simpler problems, we will use the methodology described in the next section. In a further section, we will explain the graphical notations used to represent the concepts of the ER model and of the relational one. We will finally indicate the exact structure of our thesis.

2. GENERAL METHODOLOGY

As database evolution is a very complex topic, we will decompose it into simpler problems. We therefore start by choosing a restricted relational model, the Basic Relational Model and we try to build an ER model, the Kernel, whose concepts are entirely translatable into those of the Basic Relational Model. Once these two models defined, we can study the modifications, supporting database evolution, in a very restricted framework.

As the Kernel is however a very poor model, we will enrich it into what we call the Basic ER Model. When mapping the concepts of the Basic ER Model down to the logical level, we observe that the Basic Relational Model is here not sufficient anymore and we will thus use the complete relational model, the Rich Relational Model, instead. The above mentioned enrichment of the Kernel forces us to review the previously considered modifications.

Generally, database schemas can however not be expressed in the Basic ER Model as it is still too poor. We therefore have to introduce the Rich ER Model, which allows the most commonly used concepts and we have to study how the modifications are mapped down to the Basic ER Model.

We thus distinguish three stages:

- Impact of the Modifications of the Kernel onto the Basic Relational Model
- Impact of the Modifications of the Basic ER Model onto the Rich Relational Model
- Mapping of the Modifications of the Rich ER Model down to the Basic ER Model

2.1. IMPACT OF THE MODIFICATIONS OF THE KERNEL ONTO THE BASIC RELATIONAL MODEL

We will first describe the relation between the Kernel and the Basic Relational Model and we will then discuss the impact of the modifications of the Kernel.

The Basic Relational Model is in our case an SQL model limited to the simplest objects. It allows the following concepts only:

- tables with a primary key
- columns which may be null or not
- foreign keys referencing primary keys only
- primary keys composed by one mandatory column only
- uniqueness constraints composed by one column only

This model does thus not include triggers and check clauses for example.

We will try to find a restricted ER model, the Kernel, which has a one-to-one relation with the Basic Relational Model. That means that each schema in the Kernel must have an equivalent one in the Basic Relational Model and vice versa. In other words, each object in one of the two models must have a counterpart in the other one. The relation between the Kernel and the

Basic Relational Model is illustrated in Figure 2-1. Note that a precise definition of the Kernel will be given in the first section of chapter 3 (see page 3-1).

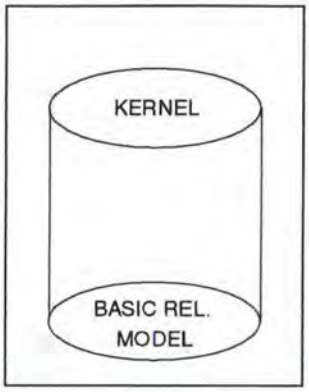


Figure 2 - 1 : Relation between the Kernel and the Basic Relational Model

Due to the one-to-one relation between the two models, the modifications made on the schemas of the Kernel are easily translatable on the schemas of the Basic Relational Model. In order to study how the modifications of the Kernel are translated, we will analyse their impact on the logical level, on the SQL database structure, on the data and on the application programs.

2.2. IMPACT OF THE MODIFICATIONS OF THE BASIC ER MODEL ONTO THE RICH RELATIONAL MODEL

We will then enrich progressively the Kernel in order to obtain an ER model, the Basic ER Model, whose concepts are fully translatable into the Rich Relational Model. By Rich Relational Model we mean a model including all possible relational concepts:

- tables
- columns which may be null or not
- foreign keys
- primary keys
- uniqueness constraints
- check constraints
- views
- indexes

  :

A definition of the Basic ER Model can be found in chapter 6 (see page 6-2). The relation between the different models seen so far is represented in Figure 2-2.

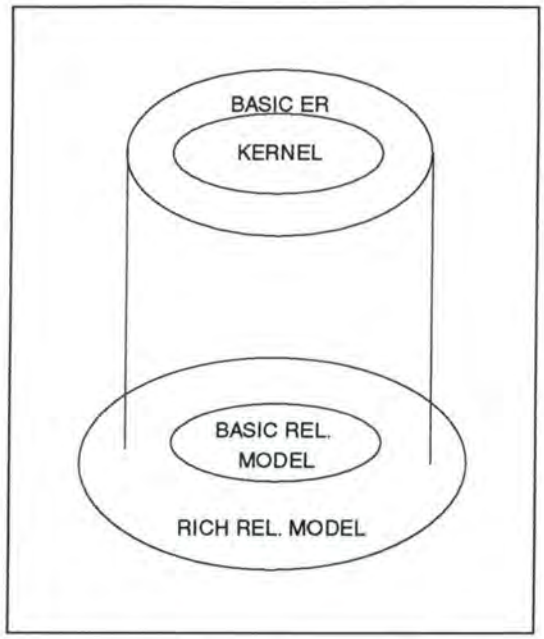


Figure 2 - 2 : Relation between the Basic ER model and the Rich Relational Model

This figure shows us that each schema expressed in the Basic ER Model is entirely translatable into the Rich Relational Model, but that the contrary is not necessarily correct. For example, certain check constraints have no equivalent in the ER models.

The previously mentioned enrichment of the Kernel forces us to review the mapping of the modifications established for the Kernel on the one hand and involves new modifications on the other hand. For example, allowing also non mono-attribute identifiers constraints us to review the modification add_unique_feature among others. We have there to pay attention to the fact that the primary key and thus the foreign keys can now be composed by several columns (see page 6-18). In addition, allowing such identifiers involves new modifications such as adding or removing an attribute to/from a unique key (see page 6-19).

2.3. Mapping of the Modifications of the Rich ER Model down to the Basic ER Model

Generally, database schemas can however not be expressed in the Basic ER Model as it is still too poor. We therefore have to introduce the Rich ER Model, which allows the most commonly used concepts (see page 7-1). Each schema expressed in this Rich ER Model must be translatable into the Basic ER Model and each modification on a Rich ER schema must have an equivalent on the Basic ER schema. This equivalent can be composed by either one or more modifications of the Basic ER Model. For example, if we want to make a compound attribute mandatory on the Rich ER Model, then on the Basic ER Model we have first to remove the coexistence constraint from the decomposed attributes and then make each of them mandatory.

The hierarchy of the different models is shown in Figure 2-3.

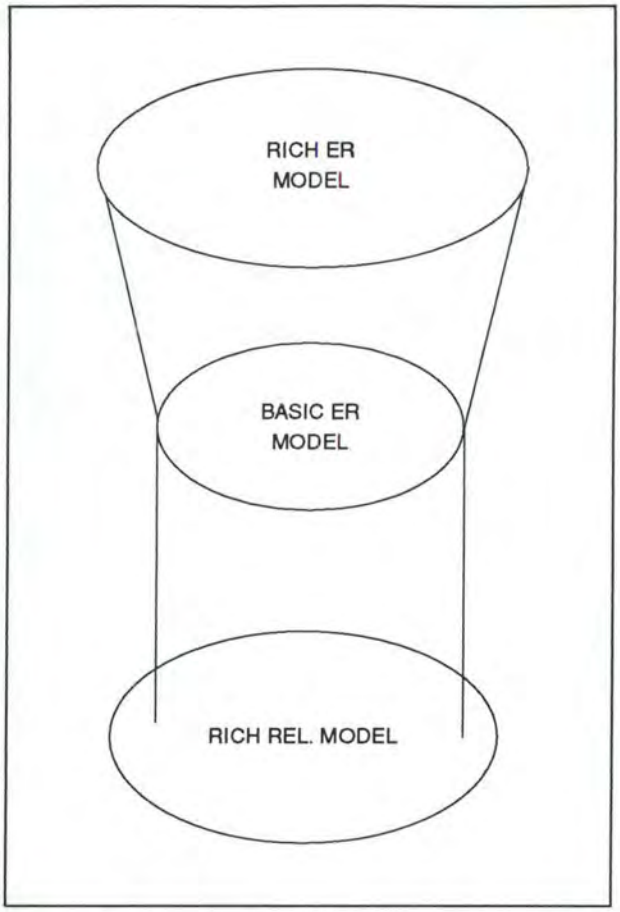


Figure 2 - 3 : The hierarchy of the different models

2.4. SUMMARY OF THE METHODOLOGY

To summarize the situation, we have established the way the modifications of the Rich ER Model are mapped down to the Rich Relational Model. In fact, each modification of the Rich ER Model is translated into one or several modification(s) of the Basic ER Model which in their turn are mapped to the Rich Relational Model. Each of the modifications of the Basic ER Model has an impact on the logical level, on the SQL database structure, on the data and on the application programs.

3. GRAPHICAL NOTATIONS

We will give the graphical notations for the conceptual and logical concepts used in our thesis. For most of the concepts of the conceptual level (see Figure 2-4), we will refer to [BOD89] and we will use a similar notation for those of the logical level (see Figure 2-5).

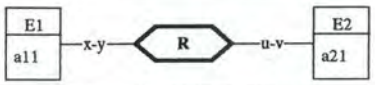
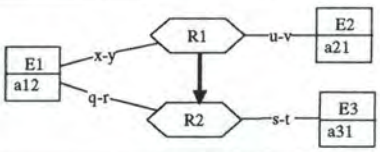
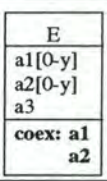
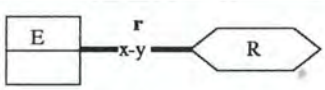
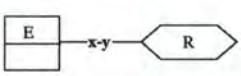
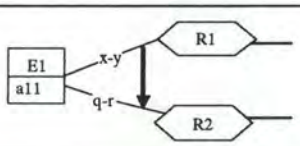
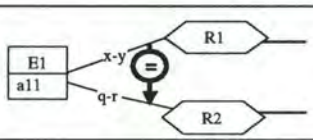
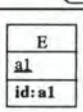
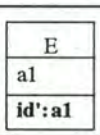
entity-type E	
relationship-type R	
functional dependency on relationship-types	
attribute a	
attribute cardinalities[1] minimum cardinality x maximum cardinality y	
coexistence constraint on attributes	
role r	
role cardinalities minimum cardinality x maximum cardinality y	
functional dependency on roles	
equality constraint on roles	
primary key	
unique key	

Figure 2 - 4 : Graphical notations for the conceptual concepts

[1] An attribute having the cardinalities [0-1] is optional
[1-1] is mandatory
[x-y] is multi-valued if y > 1

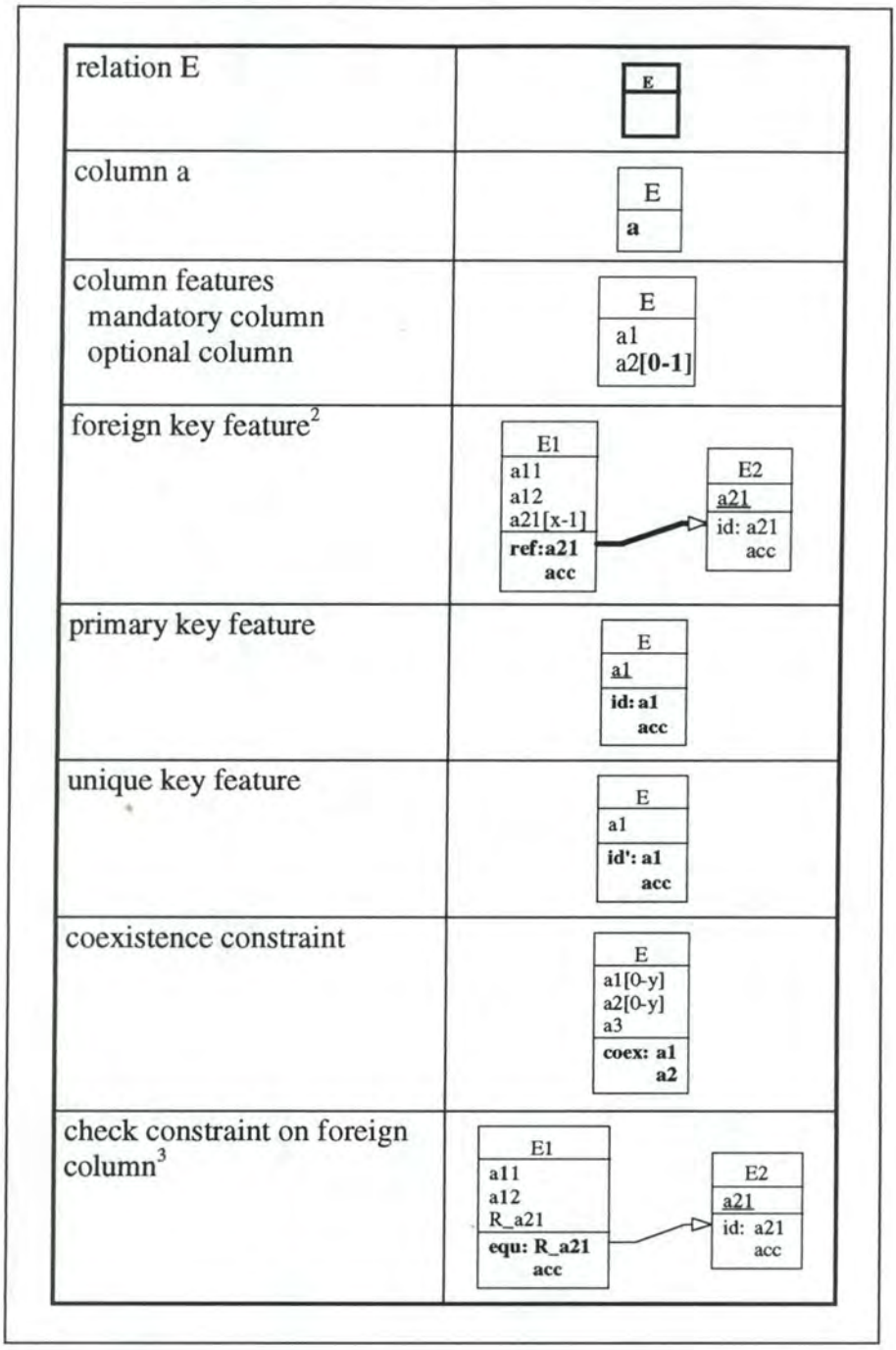


Figure 2 - 5 : Graphical notations for the logical concepts

[2] As we can see on the illustration, access keys (*acc*) will always be specified on the logical level when foreign, primary and unique key features appear. We will however not consider them in the SQL descriptions as they correspond to non-functional requirements which are not treated in our thesis.

[3] Note that *equ* represents both the foreign key feature and the check constraint.

4. STRUCTURE OF THE PROJECT

Our principle concern will be to analyze the 'Schema Modification in Relational Database Systems' on a restricted model: the Kernel. We therefore give, in the third chapter, a precise description of the Kernel and a typology of its modifications.

As our thesis is a very technical one, we have decided to adopt an inductive approach. We will thus first develop a case study and then analyse the modifications and their impact on it (see chapter 4). In a further step, we will study the modifications in general (see chapter 5). Nevertheless, as the typology of the modifications is very large, we will treat only a few of them in chapter 4 and 5 and we will discuss the totality of the modifications of the typology in appendices 1 and 2.

In the sixth chapter, we will illustrate the problems that we can expect by enlarging the Kernel to the Basic ER Model. In the seventh chapter, we will describe first the link that exists between the Rich and the Basic ER Models and then the mapping of the modifications from the Rich ER Model down to the Basic ER Model.

Finally, in the eighth chapter, we will show how these modifications could be integrated into a CASE tool. We will finish our thesis with a brief conclusion by indicating how our thesis could be continued.

Chapter 3:

Study of the Kernel

1. DESCRIPTION OF THE KERNEL

As we already said, we will define a restricted ER model, the Kernel, which has a one-to-one relation with the Basic Relational Model (for a definition see page 2-2). That means that each schema in the Kernel must have an equivalent one in the Basic Relational Model and vice versa. In other words, each object in one of the two models must have a counterpart in the other one. The relation between the Kernel and the Basic Relational Model is illustrated in Figure 3-1.

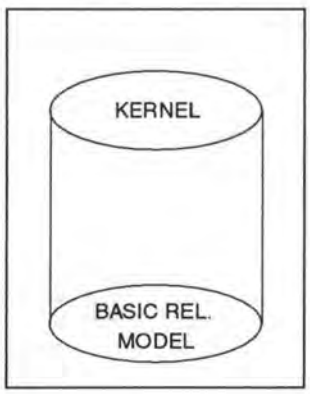


Figure 3 - 1 : Relation between the Kernel and the Basic Relational Model

We choose the Kernel as a simplified ER model which allows the following concepts only:

- *entity-types having at least one attribute and a primary key*
  If we would allow entity-types without a primary key, certain relationship-types connected to them could not be directly expressed in the relational model. We would have to add a technical identifier to the entity-types before translating these relationship-types into the relational model.

- *atomic and single-valued attributes, which can be optional or mandatory*
  This restriction is necessary as compound and/or multi-valued attributes must be decomposed and/or extracted before being translated into the relational model.

- *all the roles except those with cardinalities 1-N*
  The minimum cardinality 1 of a 1-N role could only be represented by a check constraint in the relational model and as the Basic Relational Model does not support check constraints, we have to prohibit these roles.
  In order to keep the Kernel simple and as, most of the time, the roles can be identified by their cardinalities, we do not consider names for them.

- *functional*[1] *relationship-types which are non-recursive and have the following cardinalities:*

  1-1/0-N (0-N/1-1 is symmetrical)
  0-1/0-N (0-N/0-1 is symmetrical)
  1-1/0-1 (0-1/1-1 is symmetrical)
  0-1/0-1

  We do not consider non functional relationship-types as they must be transformed into entity-types before being translated into the relational model. In addition, as we do not consider names for the roles, we cannot deal properly with recursive relationship-types.

- *identifiers of entity-types composed by one attribute only*

  We only consider identifiers of entity-types as we do not need explicit identifiers for functional relationship-types. Indeed for these relationship-types the identifer(s) is (are) derivable from their maximum cardinalities. Note that the primary keys must be composed by a mandatory attribute only.

Note:

For the moment, we only consider identifiers with one attribute as we can not express coexistence constraints in the Basic Relational Model. The problem is illustrated by Figure 3-2.

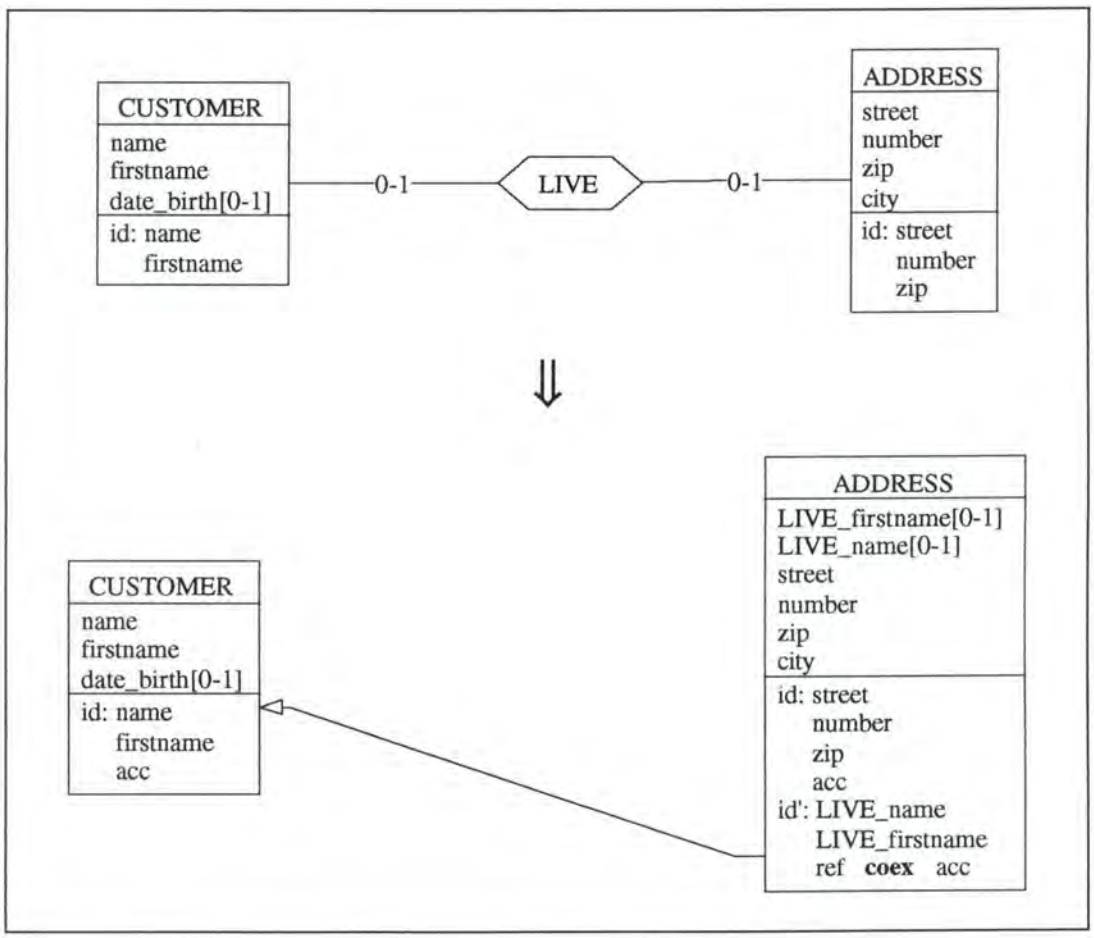


Figure 3 - 2 : An example of an identifier which is not allowed in the Kernel

[1] A relationship-type is functional if it is binary, if it has no attributes and if it is not N-N.

The coexistence constraint between LIVE_name and LIVE_firstname cannot be expressed in the Basic Relational Model.

2. TYPOLOGY OF THE MODIFICATIONS

The typology of the modifications of the Kernel is established according to two criteria: the change in semantics and the objects on which the modifications apply.

semantics → objects ↓	augmenting	decreasing	preserving
entity-type	group E_A	group E_D	group E_P
rel-type	group R_A	group R_D	group R_P
role	group Ro_A	group Ro_D	group Ro_P
attribute	group A_A	group A_D	group A_P
identifier	group Id_A	group Id_D	group Id_P

Figure 3 - 3 : Typology of the modifications

The concept 'semantics' can be understood in two different ways:

1. A first approach assimilates 'semantics' to the number of constraints expressed in a schema. For example, decreasing the maximum cardinality of a role from N to 1 would correspond to an augmentation of the semantics as we add the constraint 'max card = 1'.

2. The other approach links 'semantics' to the power of representing objects of the real world and thus to the quantity of data that the schema allows to store. For example, decreasing the maximum cardinality of a role from N to 1 would correspond in this approach to a decrease of the semantics as we could represent less.

In our thesis we choose the second approach as it seems more natural to us.

For each group of the previous table, we will consider the following modifications:

group E_A: add_entity-type

group E_D: remove_entity-type

group E_P: rename_entity-type

group R_A: add_1-1/0-1_rel-type
add_0-1/0-1_rel-type
add_1-1/0-N_rel-type
add_0-1/0-N_rel-type

group R_D: remove_1-1/0-1_rel-type
remove_0-1/0-1_rel-type
remove_1-1/0-N_rel-type
remove_0-1/0-N_rel-type

group R_P: rename_1-1/0-1_rel-type
rename_0-1/0-1_rel-type
rename_1-1/0-N_rel-type
rename_0-1/0-N_rel-type

group Ro_A: augment_max_card
decrease_min_card

group Ro_D: decrease_max_card
augment_min_card

group Ro_P: /

Note:
As we only consider binary, non-recursive relationship-types for the moment, adding or removing a role does not make any sense. In addition, we do not give a name to a role and we thus do not consider the corresponding modifications: adding or removing a name to/from a role or renaming it.

group A_A: add_optional_attribute
add_mandatory_attribute
make_attr_optional
extend_domain_attribute
change_type_int_char
change_type_float_char
change_type_date_char
change_type_date_int
change_type_int_float
change_type_date_float

group A_D: remove_optional_attribute
remove_mandatory_attribute
make_attr_mandatory
restrict_domain_attribute
change_type_char_int
change_type_float_int
change_type_char_float
change_type_char_date
change_type_int_date
change_type_float_date

group A_P: rename_optional_attribute
rename_mandatory_attribute

Note:
The operation change_type_attribute, as a whole, cannot be classified along criterium 'change in semantics' and must therefore be divided into more detailed modifications. In order to reduce the number of modifications, we have only considered four basic data types: char, integer, float and date.

group Id_A: remove_unique_feature

group Id_D: add_unique_feature

group Id_P: switch_PK_unique

Note that we do not consider semantics preserving modifications such as 'transform_1-1/0-N_rel-type→entity-type' since they generate a Kernel incompatible schema or they cannot be translated into the Basic Relational Model. We thus cannot consider such modifications because of the limitations of the two models. As soon as the limitations of the models allow it, we have however to study them too. An illustration of such a modification on the Basic ER Model is given in chapter 7 (see page 7-12).

Chapter 4:

Study of the Modifications: Case Study Approach

1. INTRODUCTION

We have now described the Kernel and we have given the typology of the possible modifications. Before studying in detail those modifications in general, we will use a case study illustrating the problems which can occur. Thus we will first describe the case study and then apply the modifications on it.

As we will not apply all the modifications on it, we will give once again the typology of the modifications, indicating this time in bold those that we will analyse in detail in this chapter. The modifications that will not be treated here can be found in appendix 1.

Modifications of the entity-types:	add_entity-type remove_entity-type **rename_entity-type**
Modifications of the relationship-types:	add_1-1/0-1_rel-type add_0-1/0-1_rel-type add_1-1/0-N_rel-type add_0-1/0-N_rel-type remove_1-1/0-1_rel-type **remove_0-1/0-1_rel-type** remove_1-1/0-N_rel-type remove_0-1/0-N_rel-type rename_1-1/0-1_rel-type rename_0-1/0-1_rel-type rename_1-1/0-N_rel-type rename_0-1/0-N_rel-type
Modifications of the roles:	**augment_max_card** decrease_min_card decrease_max_card augment_min_card
Modifications of the attributes:	add_optional_attribute add_mandatory_attribute make_attr_optional extend_domain_attribute change_type_int_char change_type_float_char change_type_date_char change_type_date_int change_type_int_float change_type_date_float remove_optional_attribute remove_mandatory_attribute **make_attr_mandatory**

	restrict_domain_attribute change_type_char_int change_type_float_int change_type_char_float change_type_char_date change_type_int_date change_type_float_date rename_optional_attribute rename_mandatory_attribute
Modifications of the identifiers:	remove_unique_feature add_unique_feature **switch_PK_unique**

2. DESCRIPTION OF THE CASE STUDY

2.1. INTRODUCTION

In order to study the impact of the modifications of the conceptual schema onto the logical schema, onto the SQL database structure, onto the data and onto the application programs, we have developed the case study described here below. This case study deals with CUSTOMERs who PLACE ORDERs which are COMPOSEd of LINEs SPECIFYing a PRODUCT.

We begin with a description of the conceptual and logical case study schemas. We then give the corresponding SQL description and a possible population of the database. Finally, we have chosen a set of program extracts, which seem interesting to be studied in our thesis.

2.2. CONCEPTUAL SCHEMA

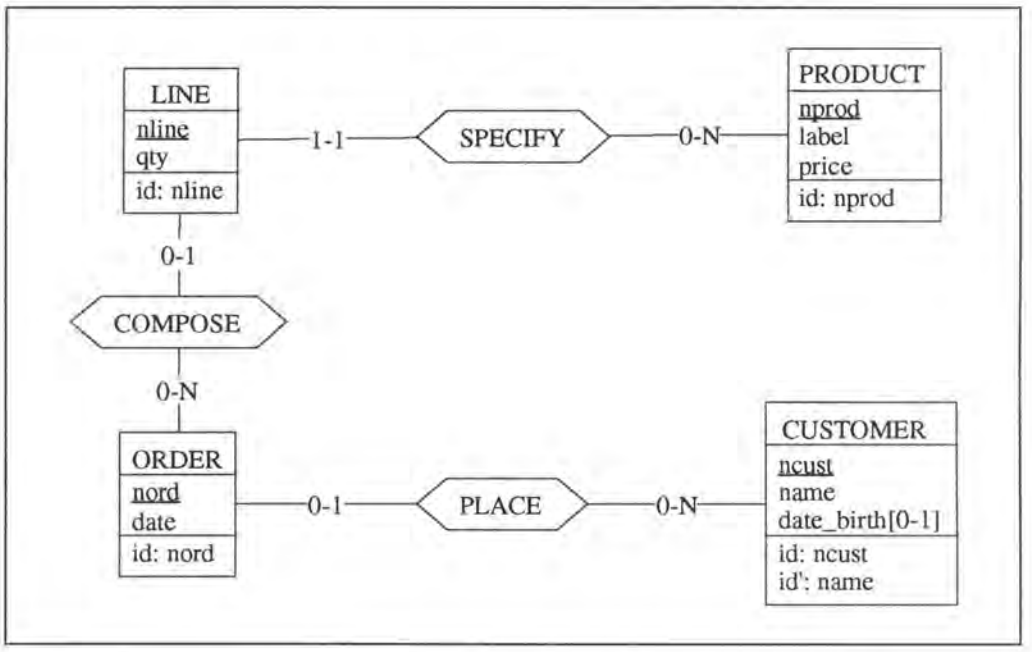


Figure 4 - 1 : Conceptual schema of the case study

2.3. LOGICAL SCHEMA

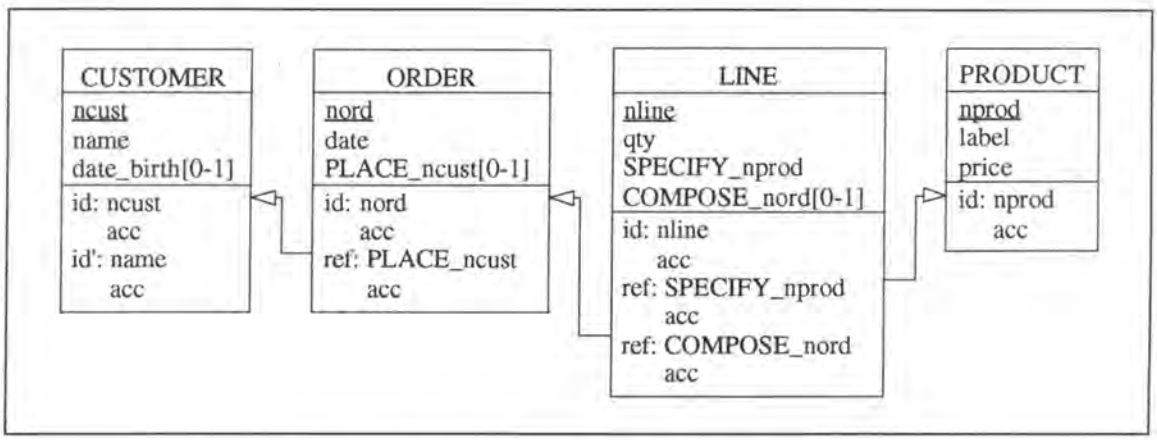


Figure 4 - 2 : Logical schema of the case study

2.4. SQL DESCRIPTION

As the syntax of the different SQL languages (DB2, RDB, ...) varies strongly with the languages, we have chosen SQL-RDB as main SQL language for our thesis. We will however sometimes indicate alternatives in other SQL languages.

```
create table CUSTOMER
      ( ncust           char(4)     not null    constraint C_ncust,
        name            char(12)    not null    constraint C_name,
        date_birth      date,
        primary key (ncust) constraint idCUS1,
        unique (name) constraint idCUS2 );


create table ORDER
      ( nord            char(4)     not null    constraint O_nord,
        date            date        not null    constraint O_date,
        PLACE_ncust     char(4),
        primary key (nord) constraint idORD1,
        foreign key (PLACE_ncust) references CUSTOMER constraint CUS1);


create table PRODUCT
      ( nprod           char(5)     not null    constraint P_nprod,
        label           char(20)    not null    constraint P_label,
        price           integer     not null    constraint P_price,
        primary key (nprod) constraint idPRO1);


create table LINE
      ( nline           char(6)     not null    constraint L_nline,
        COMPOSE_nord    char(4),
        SPECIFY_nprod   char(5)     not null    constraint L_SPECIFY_nprod,
        qty             integer     not null    constraint L_qty,
        primary key (nline) constraint idLIN1,
        foreign key (COMPOSE_nord) references ORDER constraint ORD1,
        foreign key (SPECIFY_nprod) references PRODUCT constraint PRO1);
```

2.5. DATA

CUSTOMER		
ncust	name	(date_birth)
A101	Bootsma H.	12/07/1969
D308	Ford H.	null
B234	Peiffer M.	22/06/1917
A958	Huntington G.	31/01/1969
D365	McGaw J.	29/02/1980
B472	Hasselhoff S.	null
C385	Casci G.	null
A590	Nutbush M.	09/06/1969
B253	Whopper J.	null
C395	Osborn M.	28/11/1972

Figure 4 - 3 : Table CUSTOMER

ORDER		
nord	(PLACE_ncust)	date
E386	A958	02/01/1995
F285	B472	12/03/1994
G274	null	15/07/1993
F842	C395	31/12/1994
E345	B234	05/01/1995
G222	A958	23/05/1994
F902	D365	16/09/1994
E583	B472	12/01/1995
F676	A101	26/02/1993
G809	null	23/05/1994

ORDER.PLACE_ncust in CUSTOMER.ncust

Figure 4 - 4 : Table ORDER

LINE			
nline	(COMPOSE_nord)	qty	SPECIFY_nprod
AB1234	E386	1000	AA110
GH2345	null	1518	CA510
RT3456	F285	345	AA110
ZU4567	G274	2536	BE072
ER5678	null	4587	EG880
NM6789	G274	5558	WN592
OP7890	G274	5458	RK560
JK0987	F842	5473	SW226
TZ9876	E386	623	LS906
KJ8765	null	4587	SG953
WQ7654	F902	6325	BY907
XY6543	null	9658	BY907
DS5432	E583	5698	EG880
BV1357	F842	7458	AB099
IO2468	G809	4125	AB099

LINE.COMPOSE_nord in ORDER.nord
LINE.SPECIFY_nprod in PRODUCT.nprod

Figure 4 - 5 : Table LINE

PRODUCT		
nprod	label	price
AA110	christmas tree	35
CA510	glass	50
AB099	pencil	10
BE072	gearbox	1000
WN592	wheel	850
SW226	alarm-clock	75
LS906	poster	60
SG953	toothbrush	13
EG880	postcard	9
EZ268	jumper	50
QA513	socks	23
PS375	trousers	46
RK560	mirror	93
BY905	book	186
BY907	computer	2335

Figure 4 - 6 : Table PRODUCT

2.6. PROGRAM EXTRACTS

The program extracts which are the most concerned by the modifications are those treating SQL queries. In order to obtain a significant set of these queries, we refer to the relational algebra.

The relational algebra operations are usually divided into two groups. A first group includes set operations from the mathematical set theory; these are applicable because each relation is defined to be a set of tuples. Set operations include UNION, INTERSECTION, DIFFERENCE and CARTESIAN PRODUCT. The other group consists of operations developed specifically for relational databases; these include SELECT, PROJECT and JOIN, among others.[ELM94, page 153] We will concentrate our efforts essentially on the second kind of operations but we will also describe an example of the first group of operations.

We thus consider SQL queries for the following operations: SELECT, PROJECT, JOIN and UNION.

Note:
We would have to consider the impact on other SQL commands such as 'insert into', 'update', and so on, but we will concentrate our efforts on the queries only as they are more often used.

2.6.1. Select

We will illustrate the SELECT operations by two examples:

- ```
  select *
    from CUSTOMER
    where date_birth = 09/06/1969
  ```

This query selects the number, the name and the date of birth of the CUSTOMERs who are born on the 9th of June 1969 and its result applied on the case study database is:

ncust	name	(date_birth)
A590	Nutbush M.	09/06/1969

- ```
  select *
    from CUSTOMER
    where ncust in (select PLACE_ncust
                      from ORDER
                      where nord in (select COMPOSE_nord
                                       from LINE
                                       where SPECIFY_nprod = 'AA110'))
  ```

This query selects all the information of the CUSTOMERs who have ORDERed the PRODUCT 'AA110' and it has as result:

ncust	name	(date_birth)
A958	Huntington G.	31/01/1969
B472	Hasselhoff S.	null

2.6.2. Project

```
select nline, COMPOSE_nord
  from LINE
  where SPECIFY_nprod = WN592
```

This query selects nline and COMPOSE_nord of the rows of table LINE, which reference PRODUCT WN592 and it has as result:

nline	(COMPOSE_nord)
NM6789	G274

2.6.3. Join

```
select name, nord
  from CUSTOMER, ORDER
  where (ncust = PLACE_ncust) and (date_birth < 01/01/1977)
```

This query selects the name of the CUSTOMERs and the order number of the ORDERs placed by the CUSTOMERs who are born before the 1st of January 1977 and it has as result:

name	nord
Bootsma H.	F676
Huntington G.	G222
Peiffer M.	E345
Huntington G.	E386
Osborn M.	F842

2.6.4. Union

```
select SPECIFY_nprod
  from LINE
  where qty > 4000
UNION
select nprod
  from PRODUCT
  where price <= 50
```

This query selects the product number of the PRODUCTs which cost less than 50 or which have been ordered more than 4000 times in a single LINE. The result of this query is:

nprod
EG880
WN592
RK560
SW226
SG953
BY907
AB099
AA110
CA510
EZ268
QA513
PS375

3. STUDY OF THE MODIFICATIONS: CASE STUDY APPROACH

3.1. INTRODUCTION

In the remaining of this chapter, we have to study the modifications of the conceptual level and their impact on the logical level, on the SQL database structure, on the data and on the application programs. This impact is shown in Figure 4-7.

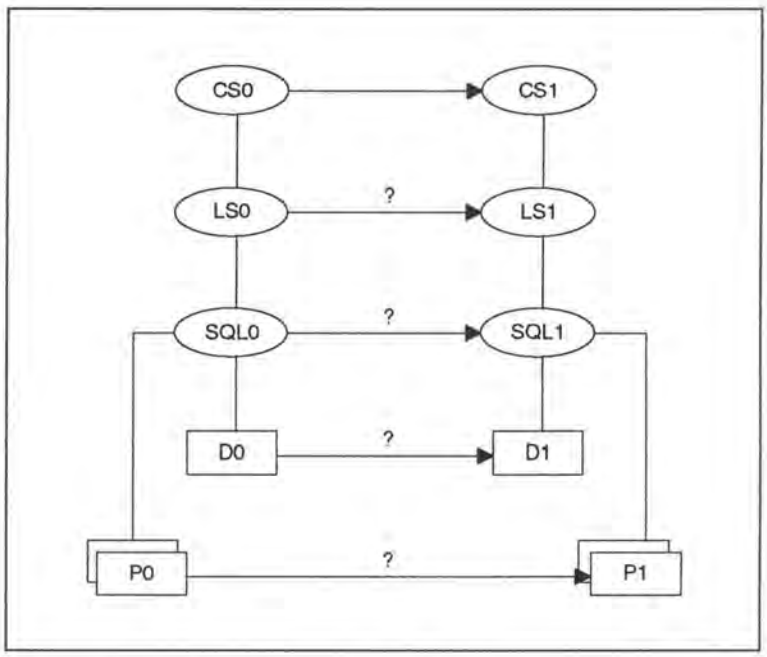


Figure 4 - 7 : Representation of the database evolution problem

If the conceptual schema CS0 has been changed, the logical schema LS0 and the SQL description SQL0 must be changed accordingly. Data D0 is no longer valid and has to be converted into data D1. Finally, the application programs P0 must be partly rewritten in order to comply with the new data structures described in SQL1. [HAI94a] Note that they must be manually changed as each application must be treated separately, according to its own logic. The best help we can supply is to indicate the concerned program sections and to give hints on what kind of changes to be performed.

As shown in the third chapter, the modifications are classified according to the objects on which they apply on the one hand and, on the other hand, according to their nature: augmenting, decreasing or preserving semantics (see page 3-4).

For each object, we will analyse only one modification. The analysis will be decomposed into two parts: its classification and its description. The description part is essentially divided into

three subparts: the impact of the modification on the Logical Schema, on the SQL Description & Data and on the Program Extracts.

3.2. RENAME_ENTITY-TYPE[1]

3.2.1. Classification of the Modification

As shown in Figure 4-8, rename_entity-type is a modification on entity-types which preserves the semantics.

semantics → / objects ↓	augmenting	decreasing	preserving
entity-type			X
rel-type			
role			
attribute			
identifier			

Figure 4 - 8 : Classification of rename_entity-type

3.2.2. Description of the Modification

Let us suppose we want to change, in our case study, the entity-type CUSTOMER into CLIENT.

[1]Normally we would have to add the following precondition: 'The new name of the entity-type that should be renamed must not yet exist.' As such preconditions are trivial, we will not indicate them anymore.

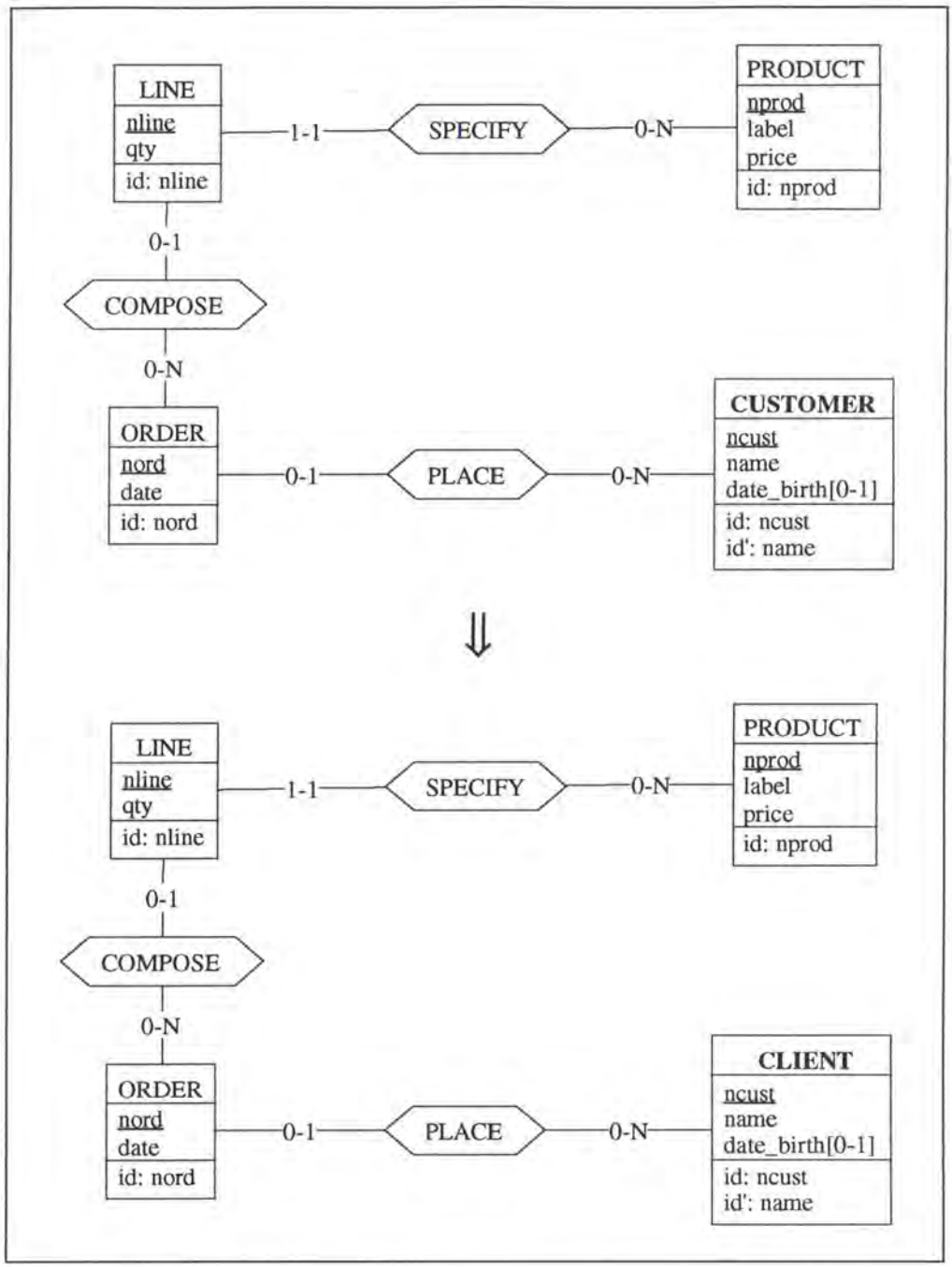


Figure 4 - 9 : Renaming an entity-type on the conceptual level

3.2.2.1. Logical Schema

In the logical schema, we have to change the name of the corresponding relation.

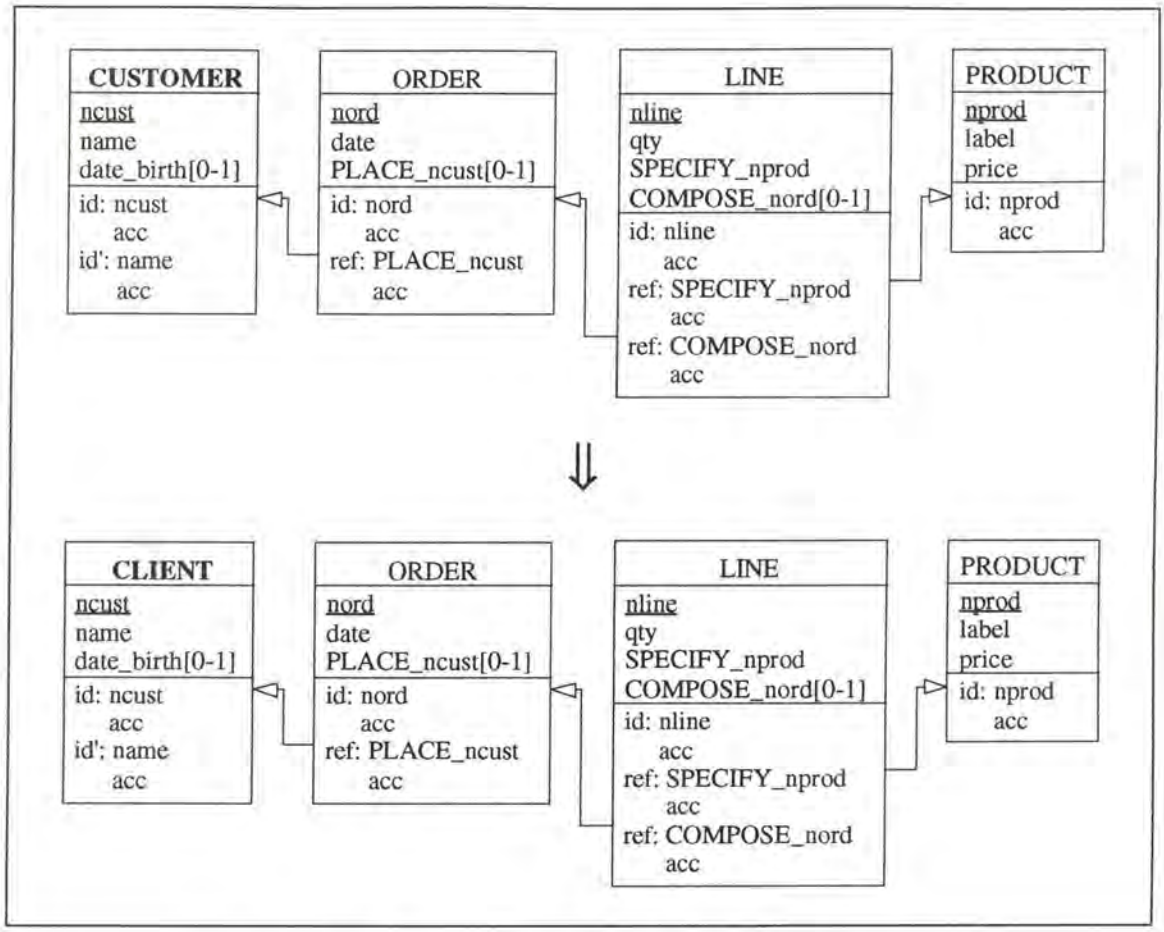


Figure 4 - 10 : Renaming an entity-type on the logical level

3.2.2.2. SQL Description & Data

In some SQL languages there may be a 'rename table' command. The modification would thus become:

```
alter table CUSTOMER
  rename table CLIENT on cascade;
```

In SQL-RDB however, no such command exists and we have therefore to create a new table and to copy the data into it.

```
create table CLIENT
        ( ncust          char(4)     not null    constraint C_ncust,
          name           char(12)    not null    constraint C_name,
          date_birth     date,
          primary key (ncust)  constraint idCLI1,
          unique (name)  constraint idCLI2 );
insert into CLIENT
  select ncust, name, date_birth
  from CUSTOMER;
alter table ORDER
  drop constraint CUS1,
  add constraint foreign key (PLACE_ncust) references CLIENT
                                                  constraint CLI1;
```

```
(* For each view defined on table CUSTOMER, we have to redefine it on table
CLIENT. In future we will not consider views anymore as they do not
correspond to ER objects. *)

drop table CUSTOMER cascade;
```

No data is lost as the data is just moved from one table into another.

Notes:

- This operation in SQL-RDB is often a very slow one as we have to copy a whole table. We thus recommend to create a view CLIENT which includes only the table CUSTOMER. This could be realised by the following command:

```
create view CLIENT
  as select *
     from CUSTOMER
```

- Other SQL languages, such as DB2, offer another possibility to implement the modification: giving a synonym to the entity-type (that has to be renamed) instead of renaming it properly. This alternative could be realised by the following SQL command:

```
create synonym CLIENT
  for CUSTOMER;
```

Note that in both cases the original table is however not renamed.

3.2.2.3. Program Extracts

In all the select queries referencing CUSTOMER, we have to rename it with CLIENT. For example, the first select query of our case study (see page 4-7) would become:

```
select *
  from CLIENT
  where date_birth = 09/06/1969
```

In addition to the select queries, we have also to review the application programs in which they appear. For instance, we must rename certain variables and/or some fields or headings in the user interfaces. Finally, let us note that the documentation should also be updated.

3.3. REMOVE_0-1/0-1_REL-TYPE

3.3.1. Classification of the Modification

As shown in Figure 4-11, remove_0-1/0-1_rel-type is a modification on relationship-types which decreases the semantics.

semantics → / objects ↓	augmenting	decreasing	preserving
entity-type			
rel-type		X	
role			
attribute			
identifier			

Figure 4 - 11 : Classification of remove_0-1/0-1_rel-type

3.3.2. Description of the Modification

Let us suppose that we want to remove the 0-1/0-1 relationship-type WORK between ADDRESS and CUSTOMER.

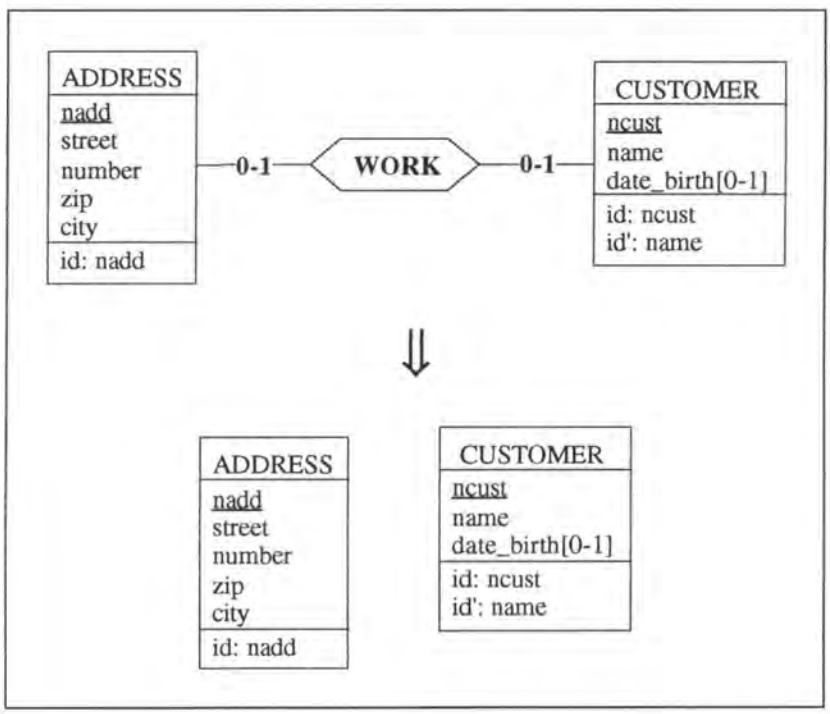


Figure 4 - 12 : Removing a 0-1/0-1 relationship-type on the conceptual level

We have to consider two possible implementations for the relationship-type WORK:

- WORK is implemented by a foreign key in ADDRESS
- WORK is implemented by a foreign key in CUSTOMER

3.3.2.1. WORK is implemented by a foreign key in ADDRESS

3.3.2.1.1. Logical Schema

We remove the column WORK_ncust from ADDRESS with its candidate and foreign key features.

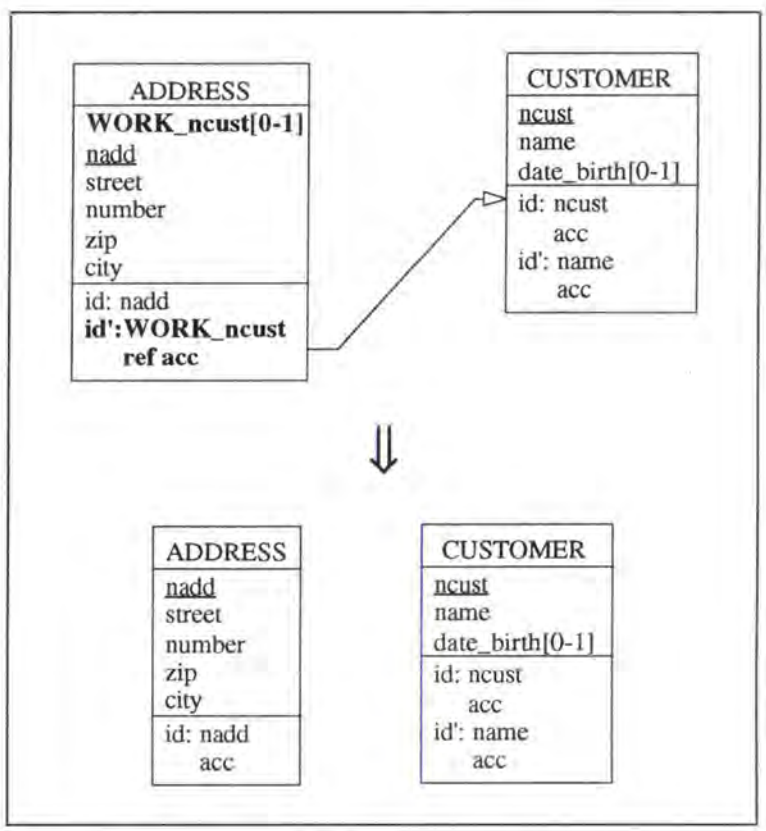


Figure 4 - 13 : Removing a 0-1/0-1 relationship-type on the logical level

3.3.2.1.2. SQL Description & Data

```
alter table ADDRESS
  drop constraint idADD2,      (* we remove the unique key feature *)
  drop constraint CUS1,        (* we remove the foreign key feature *)
  drop WORK_ncust;
```

The link between a CUSTOMER and his/her ADDRESS is lost.

3.3.2.1.3. Program Extracts

All the select queries which reference WORK_ncust in ADDRESS must be modified. Depending on their context, the user has either to drop the select queries or to modify them for example as follows:

```
select name, street, number, zip, city
  from ADDRESS, CUSTOMER
  where (WORK_ncust = ncust) and
        (ncust in (select PLACE_ncust
                   from ORDER
                   where nord in (select COMPOSE_nord
                                  from LINE
                                  where SPECIFY_nprod = 'AA110')))
```

```
select name
  from CUSTOMER
  where ncust in ( select PLACE_ncust
                   from ORDER
                   where nord in ( select COMPOSE_nord
                                   from LINE
                                   where SPECIFY_nprod = 'AA110' ))
```

The application programs in which these queries appear must also be reviewed. We cannot describe a general method how to deal with these application programs as each one of them must be treated individually. Let us for example consider a screen which displays the information about a CUSTOMER, including his/her working ADDRESS. As we have now lost the link between a CUSTOMER and his/her working ADDRESS, the user has to decide what should happen to the part of the screen allocated to the working ADDRESS. He can either drop it and rearrange the screen or reuse it for another purpose (for example: for indicating the living ADDRESS of the CUSTOMER). In addition, the user has to check whether the variables are still all needed and he has also to update the documentation.

3.3.2.2. WORK is implemented by a foreign key in CUSTOMER

3.3.2.2.1. Logical Schema

We remove the column WORK_nadd in CUSTOMER with its candidate and foreign key features.

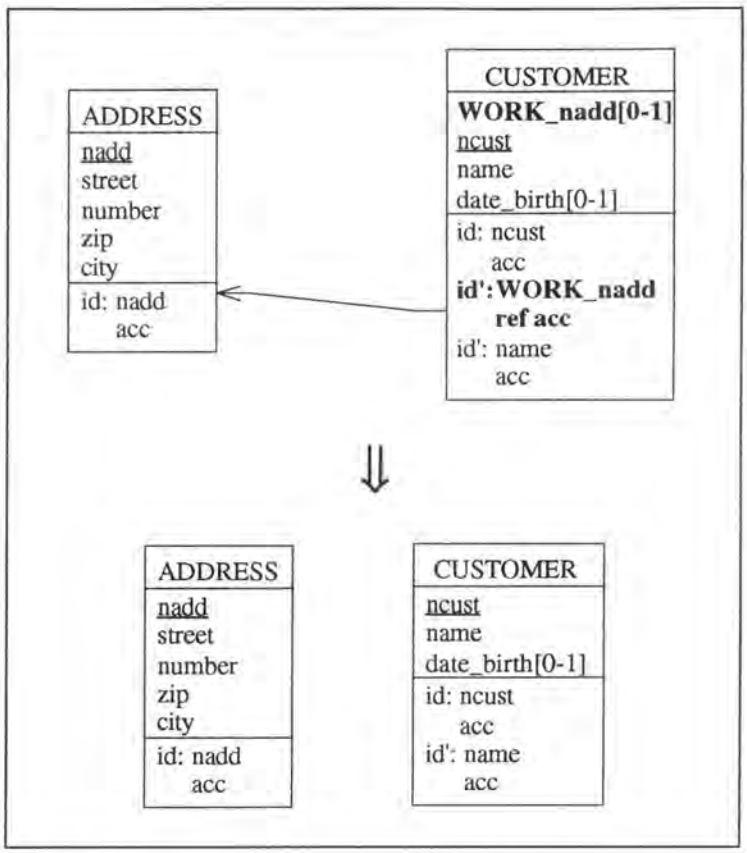


Figure 4 - 14 : Removing a 0-1/0-1 relationship-type on the logical level

3.3.2.2.2. SQL Description & Data

```
alter table CUSTOMER
  drop constraint idCUS2,      (* we remove the unique key feature *)
  drop constraint ADD1,        (* we remove the foreign key feature *)
  drop WORK_nadd;
```

The link between a CUSTOMER and his/her ADDRESS is lost.

3.3.2.2.3. Program Extracts

The impact on the program extracts are similar to those of the previous case (see page 4-16).

3.4. AUGMENT_MAX_CARD

3.4.1. Classification of the Modification

As shown in Figure 4-15, augment_max_card is a modification on roles which augments the semantics.

semantics → / objects ↓	augmenting	decreasing	preserving
entity-type			
rel-type			
role	X		
attribute			
identifier			

Figure 4 - 15 : Classification of augment_max_card

3.4.2. Description of the Modification

Precondition:
Given the restrictions of the relationship-types in the Kernel (see page 3-2) the only augmentations of the maximum cardinality of a role that we accept so far are:

- 1-1/0-1 → 1-1/0-N
- 0-1/0-1 → 0-1/0-N

We consider an example for each of the two cases.

3.4.2.1. 1-1/0-1 → 1-1/0-N

Let us consider the example where a CUSTOMER LIVEs at an ADDRESS. We want to augment to N the maximum cardinality of the 0-1 role of relationship-type LIVE.

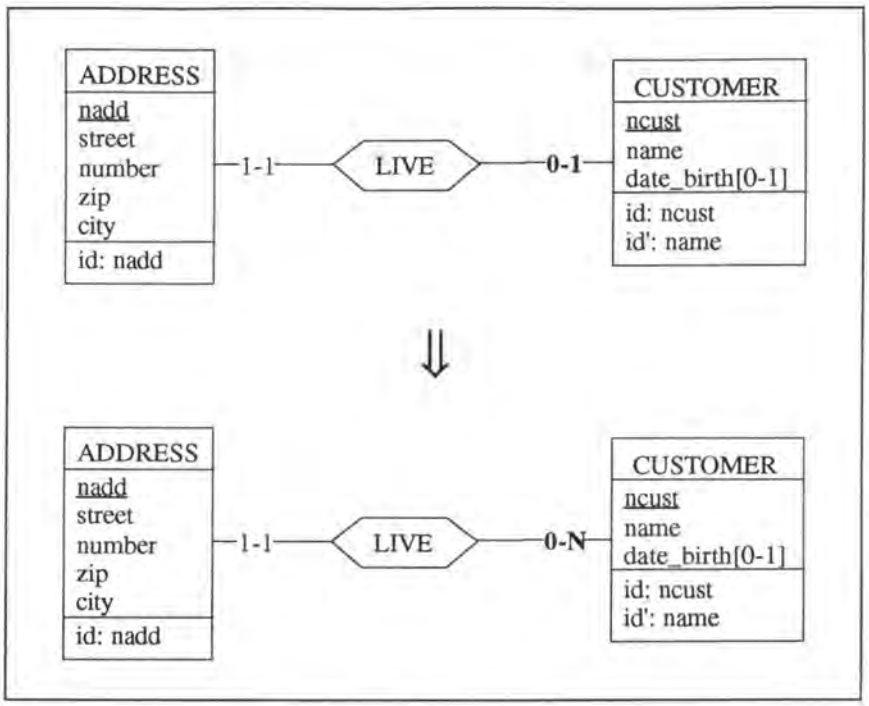


Figure 4 - 16 : Augmenting the maximum cardinality of a role to N in an 1-1/0-1 relationship-type on the conceptual level

3.4.2.1.1. Logical Schema

We have to remove the candidate key feature from the foreign key LIVE_ncust in relation ADDRESS.

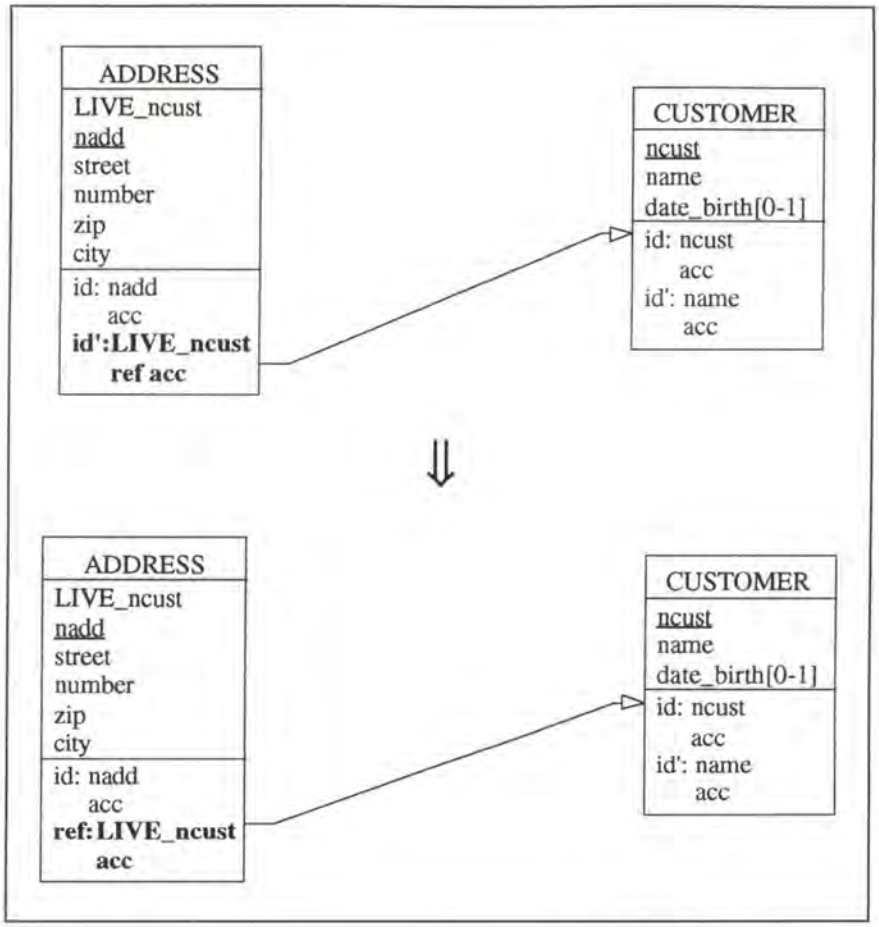


Figure 4 - 17 : Augmenting the maximum cardinality of a role to N in an 1-1/0-1 relationship-type on the logical level

3.4.2.1.2. SQL Description & Data

```
alter table ADDRESS
  drop constraint idADD2;     (* we remove the unique key feature *)
```

No modifications are made on the data.

3.4.2.1.3. Program Extracts

Let us consider the following program extract:

```
var street: STRING[20];
    number: INTEGER;
    zip: INTEGER;
    city: STRING[20];

:
exec SQL
      select street, number, zip, city
        into :street, :number, :zip, :city
        from ADDRESS
        where LIVE_ncust = 'A101'
end exec;
if SQLCODE = 0          (* if such a row has been found *)
then begin
```

```
        write(street);
        write(number);
        write(zip);
        write(city);
      end;
:
```

We have to adapt this extract, as shown here below, in order to allow a CUSTOMER to have several ADDRESSes. Note that the simple treatment (if...then) has to be replaced by a loop treatment (while...do).

```
var street: STRING[20];
    number: INTEGER;
    zip: INTEGER;
    city: STRING[20];

   :
exec SQL
      declare c cursor for
        select street, number, zip, city
          from ADDRESS
          where LIVE_ncust = 'A101';
      open c;
      fetch c into :street, :number, :zip, :city;
end exec;
while SQLCODE = 0       (* the last item has not yet been treated *)
  do begin
        write (street);
        write (number);
        write (zip);
        write (city);
        exec SQL
            fetch c into :street, :number, :zip, :city;
        end exec
     end;
exec SQL
      close c
end exec;
   :
```

In addition, let us reconsider the screen which displays the information about a CUSTOMER, including his/her working ADDRESS. As a CUSTOMER can now have several ADDRESSes, the user has to rearrange the screen so that it can display several ADDRESSes. Finally, the user has to replace certain variables by arrays and he has also to update the documentation.

3.4.2.2. 0-1/0-1 → 0-1/0-N

We want to transform the role of the relationship-type WORK played by CUSTOMER into 0-N.

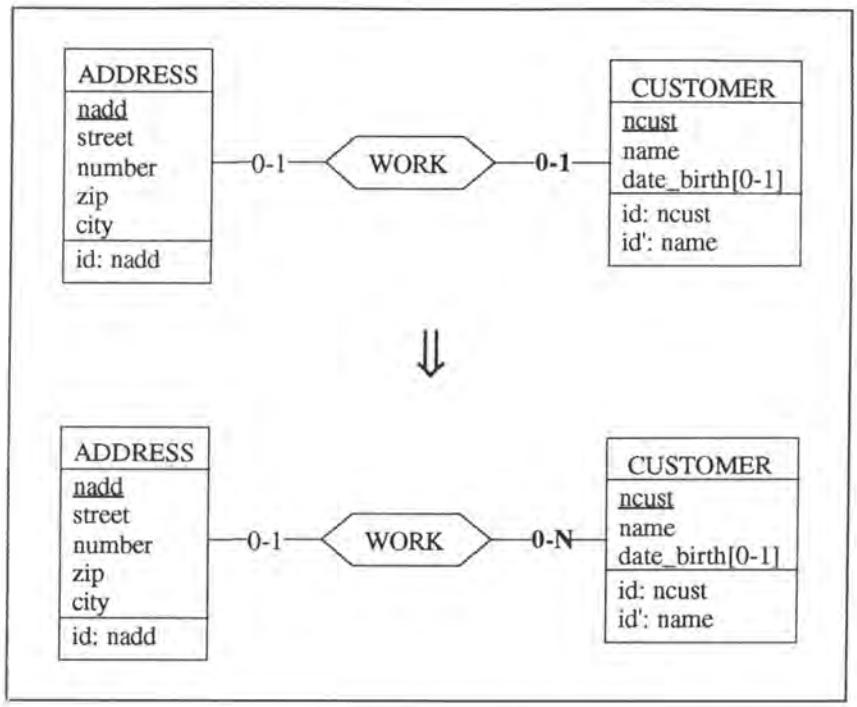


Figure 4 - 18 : Augmenting the maximum cardinality of a role to N in an 0-1/0-1 relationship-type on the conceptual level

There are two possible representations on the logical level:

- WORK is implemented by a foreign key in relation ADDRESS
- WORK is implemented by a foreign key in relation CUSTOMER

3.4.2.2.1. WORK is implemented by a foreign key in relation ADDRESS

3.4.2.2.1.1. Logical Schema

The initial logical schema is:

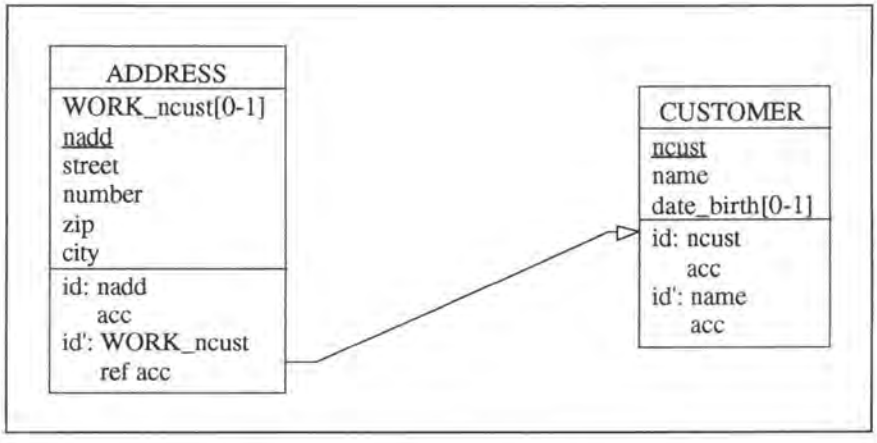


Figure 4 - 19 ; The initial logical schema

This case is identical to the case 3.4.2.1.1. (see page 4-19).

3.4.2.2.2. WORK is implemented by a foreign key in relation CUSTOMER

3.4.2.2.2.1. Logical Schema

On the logical level, the transformation is:

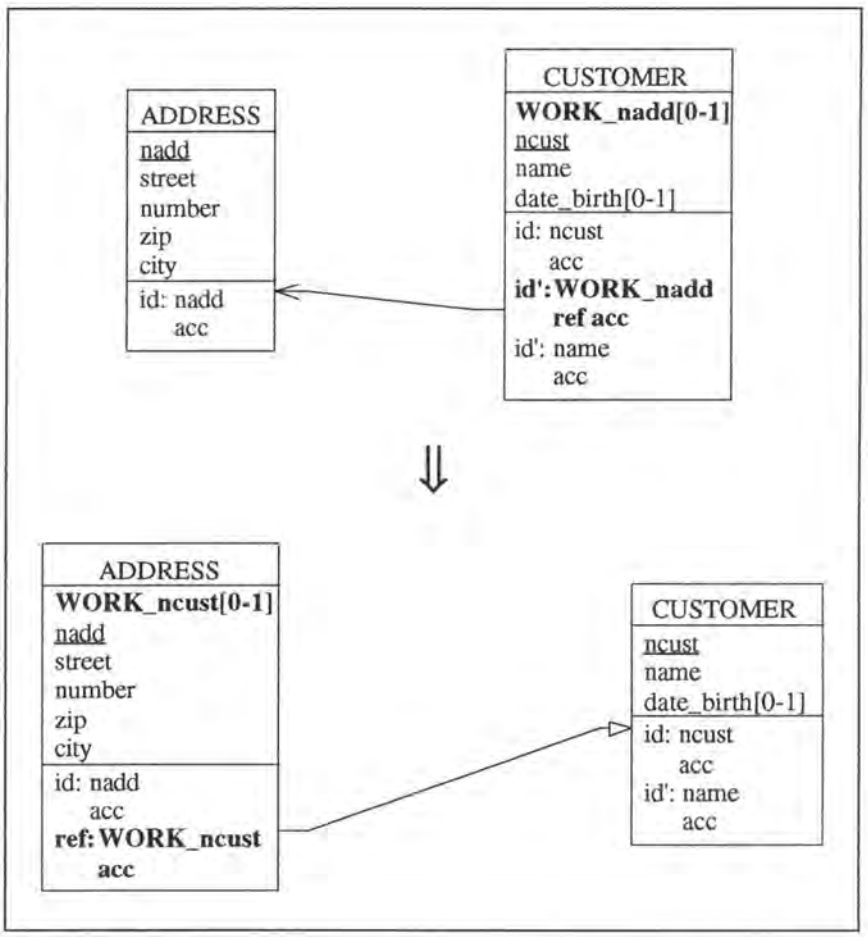


Figure 4 - 20 : Augmenting the maximum cardinality of a role to N in an 0-1/0-1 relationship-type on the logical level

3.4.2.2.2.2. SQL Description & Data

```
var cust: STRING[4];
    add: INTEGER;

exec SQL
      (* we create the new foreign key column *)
      alter table ADDRESS
        add WORK_ncust char(4);
      (* we copy the data representing relationship-type WORK from table
         CUSTOMER into table ADDRESS *)
      declare c cursor for
        select ncust, WORK_nadd
          from CUSTOMER
          where WORK_nadd is not null;
      open c;
      fetch c into :cust, :add;
end exec;
```

```
while SQLCODE = 0       (* the last item has not yet been treated *)
do begin
      exec SQL
            update ADDRESS
              set WORK_ncust = :cust
              where nadd = :add;
            fetch c into :cust, :add;
      end exec;
   end;
exec SQL
      (* we add and remove the necessary constraints *)
      alter table ADDRESS
        add constraint foreign key (WORK_ncust) references
                                                CUSTOMER constraint CUS1;
      alter table CUSTOMER
        drop constraint idCUS2,     (* we remove the unique key feature *)
        drop constraint ADD1,       (* we remove the foreign key feature *)
        drop WORK_nadd;
      close c;
end exec;
```

Note that no data is lost as the data representing relationship-type WORK is moved from table CUSTOMER into table ADDRESS.

3.4.2.2.2.3. Program Extracts

Application programs referencing the foreign key representing relationship-type WORK must be reviewed. Two possible modifications are:

-

```
var add: STRING[12];

exec SQL
      select WORK_nadd
        into :add
        from CUSTOMER
        where name = 'Hasselhoff S.';
end exec;
if SQLCODE = 0          (* if such a row has been found *)
then write (add);
```

```
var add: STRING[12];

exec SQL
      declare c cursor for
        select nadd
          from ADDRESS
          where WORK_ncust in (select ncust
                               from CUSTOMER
                               where name = 'Hasselhoff S.')
      open c;
      fetch c into :add;
end exec;
while SQLCODE = 0       (* the last item has not yet been treated *)
do begin
      write (add);
      exec SQL
            fetch c into :add;
      end exec;
   end;
exec SQL
      close c
end exec;
```

- ```
  select street, city
    from ADDRESS
    where nadd in (select WORK_nadd
                   from CUSTOMER
                   where name like '%Dupont%')
  ```

  ⇓

  ```
  select street, city
    from ADDRESS
    where WORK_ncust in (select ncust
                         from CUSTOMER
                         where name like '%Dupont%').
  ```

Concerning the application programs, similar remarks as for the case 3.4.2.1.3. (see page 4-20) can be formulted here.

3.5. MAKE_ATTR_MANDATORY

3.5.1. Classification of the Modification

As shown in Figure 4-21, make_attr_mandatory is a modification on attributes which decreases the semantics.

semantics → / objects ↓	augmenting	decreasing	preserving
entity-type			
rel-type			
role			
attribute		X	
identifier			

Figure 4 - 21 : Classification of make_attr_mandatory

3.5.2. Description of the Modification

We have to distinguish whether the attribute which we want to make mandatory is a unique key or not. We will treat first the case in which the attribute is not a unique key.

3.5.2.1. The attribute is not a unique key

Let us suppose we want to make date_birth mandatory in CUSTOMER.
  ```

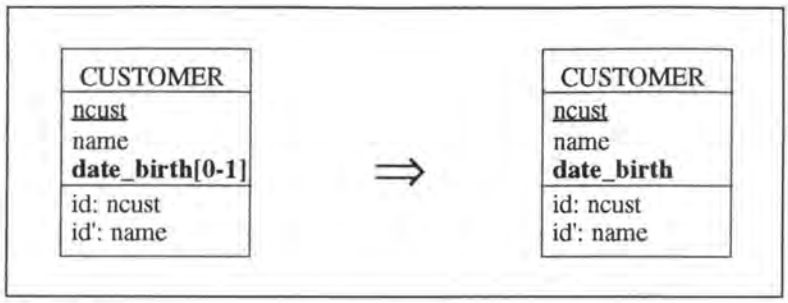


Figure 4 - 22 : Making a non-key attribute mandatory on the conceptual level

3.5.2.1.1. Logical Schema

We make date_birth mandatory in relation CUSTOMER.

3.5.2.1.2. SQL Description & Data

```
update CUSTOMER
  set date_birth = 00/00/0000
  where date_birth is null;
alter table CUSTOMER
  alter date_birth not null constraint C_date_birth;
```

This way of implementing the modification involves no loss of data as we replace the null values by a default value:

CUSTOMER		
ncust	name	(date_birth)
A101	Bootsma H.	12/07/1969
D308	Ford H.	**00/00/0000**
B234	Peiffer M.	22/06/1917
A958	Huntington G.	31/01/1969
D365	McGaw J.	29/02/1980
B472	Hasselhoff S.	**00/00/0000**
C385	Casci G.	**00/00/0000**
A590	Nutbush M.	09/06/1969
B253	Whopper J.	**00/00/0000**
C395	Osborn M.	28/11/1972

Figure 4 - 23 : Table CUSTOMER when the null values of column date_birth are replaced by a default value

We thus have the choice whether to remove or not the rows of table CUSTOMER with the default value in column date_birth. If we want to remove those rows, we can use the following operation:

```
delete
  from CUSTOMER
  where date_birth = 00/00/0000
```

We then still have to decide what should happen to the ORDERs PLACEd by these CUSTOMERs. Note that the only CUSTOMER who has PLACEd ORDERs is CUSTOMER B472. We have two choices:

A. Set PLACE_ncust to null for the ORDERs PLACEd by the CUSTOMER B472:

ORDER		
nord	(PLACE_ncust)	date
E386	A958	02/01/1995
F285	**null**	12/03/1994
G274	null	15/07/1993
F842	C395	31/12/1994
E345	B234	05/01/1995
G222	A958	23/05/1994
F902	D365	16/09/1994
E583	**null**	12/01/1995
F676	A101	26/02/1993
G809	null	23/05/1994

ORDER.PLACE_ncust in CUSTOMER.ncust

Figure 4 - 24 : Table ORDER when certain PLACE_ncust values are set to null

B. Delete also the ORDERs PLACEd by the CUSTOMER B472:

ORDER		
nord	(PLACE_ncust)	date
E386	A958	02/01/1995
G274	null	15/07/1993
F842	C395	31/12/1994
E345	B234	05/01/1995
G222	A958	23/05/1994
F902	D365	16/09/1994
F676	A101	26/02/1993
G809	null	23/05/1994

ORDER.PLACE_ncust in CUSTOMER.ncust

Figure 4 - 25 : Table ORDER when certain rows are deleted

If we have decided to delete also the ORDERs, we have finally to decide what should happen to the LINEs which COMPOSE the ORDERs E583 and F285. Here again we have the two same choices:

B1. Set COMPOSE_nord to null for the LINEs associated to the ORDERs that have been removed:

LINE			
nline	(COMPOSE_nord)	qty	SPECIFY_nprod
AB1234	E386	1000	AA110
GH2345	null	1518	CA510
RT3456	**null**	345	AA110
ZU4567	G274	2536	BE072
ER5678	null	4587	EG880
NM6789	G274	5558	WN592
OP7890	G274	5458	RK560
JK0987	F842	5473	SW226
TZ9876	E386	623	LS906
KJ8765	null	4587	SG953
WQ7654	F902	6325	BY907
XY6543	null	9658	BY907
DS5432	**null**	5698	EG880
BV1357	F842	7458	AB099
IO2468	G809	4125	AB099

LINE.COMPOSE_nord in ORDER.nord
LINE.SPECIFY_nprod in PRODUCT.nprod

Figure 4 - 26 : Table LINE where certain values for column COMPOSE_nord are set to null

B2. Delete also the LINEs associated to the ORDERs that have been removed:

LINE			
nline	(COMPOSE_nord)	qty	SPECIFY_nprod
AB1234	E386	1000	AA110
GH2345	null	1518	CA510
ZU4567	G274	2536	BE072
ER5678	null	4587	EG880
NM6789	G274	5558	WN592
OP7890	G274	5458	RK560
JK0987	F842	5473	SW226
TZ9876	E386	623	LS906
KJ8765	null	4587	SG953
WQ7654	F902	6325	BY907
XY6543	null	9658	BY907
BV1357	F842	7458	AB099
IO2468	G809	4125	AB099

LINE.COMPOSE_nord in ORDER.nord
LINE.SPECIFY_nprod in PRODUCT.nprod

Figure 4 - 27 : Table LINE where certain rows are deleted

3.5.2.1.3. Program Extracts

Select queries referencing the null value of column date_birth could be modified as follows:

```
select ncust
  from CUSTOMER
  where date_birth is null
```

```
select ncust
  from CUSTOMER
  where date_birth = 00/00/0000
```

in case we have not dropped the data and should be dropped else.
Note that all the application programs in which such queries appear must also be reviewed. For example, the tests on the null value of date_birth must be changed either by testing the default value or by simply removing them.

3.5.2.2. The attribute is a unique key

We want to make the attribute label in entity-type FACTORY mandatory.

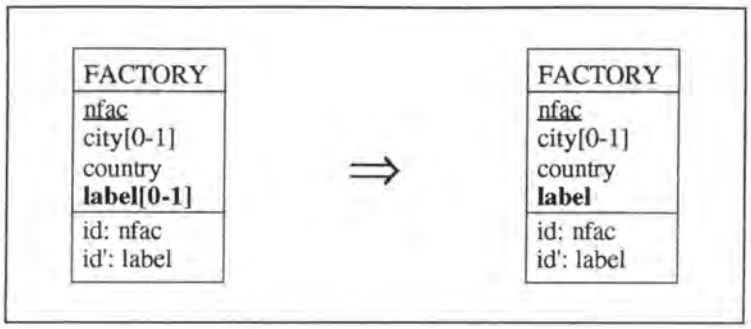


Figure 4 - 28 : Making a (unique) key attribute mandatory on the conceptual level

3.5.2.2.1. Logical Schema

We make the column label in relation FACTORY mandatory.

3.5.2.2.2. SQL Description & Data

```
(* we cannot use here a default value because of the unique key feature of
   column label *)
delete from FACTORY
  where label is null;
alter table FACTORY          (* we can only alter a column on which no
                                constraints apply *)
  drop constraint idFAC2,     (* we remove the unique key constraint *)
  alter label not null constraint F_label,
  add constraint unique(label) constraint idFAC2;
```

All the rows which had a null value for column label in table FACTORY are lost.

3.5.2.2.3. Program Extracts

All the application programs containing queries referencing the null value of column label must be reviewed in a similar way as in the previous case (see page 4-29).

3.6. SWITCH_PK_UNIQUE

3.6.1. Classification of the Modification

As shown in the Figure 4-29, switch_PK_unique is a modification on identifiers which preserves the semantics.

semantics → / objects ↓	augmenting	decreasing	preserving
entity-type			
rel-type			
role			
attribute			
identifier			X

Figure 4 - 29 : Classification of switch_PK_unique

3.6.2. Description of the Modification

This modification is used to transform a primary key into a unique key and vice versa. The user has the choice whether to specify a unique key or not. If he does not specify any unique key, then a technical identifier is created as primary key.

Precondition:
If a unique key is specified, then it must not be optional. This is due to the fact that SQL-RDB does not allow optional attributes as primary key.

The structure of the modification switch_PK_unique is represented in Figure 4-30.

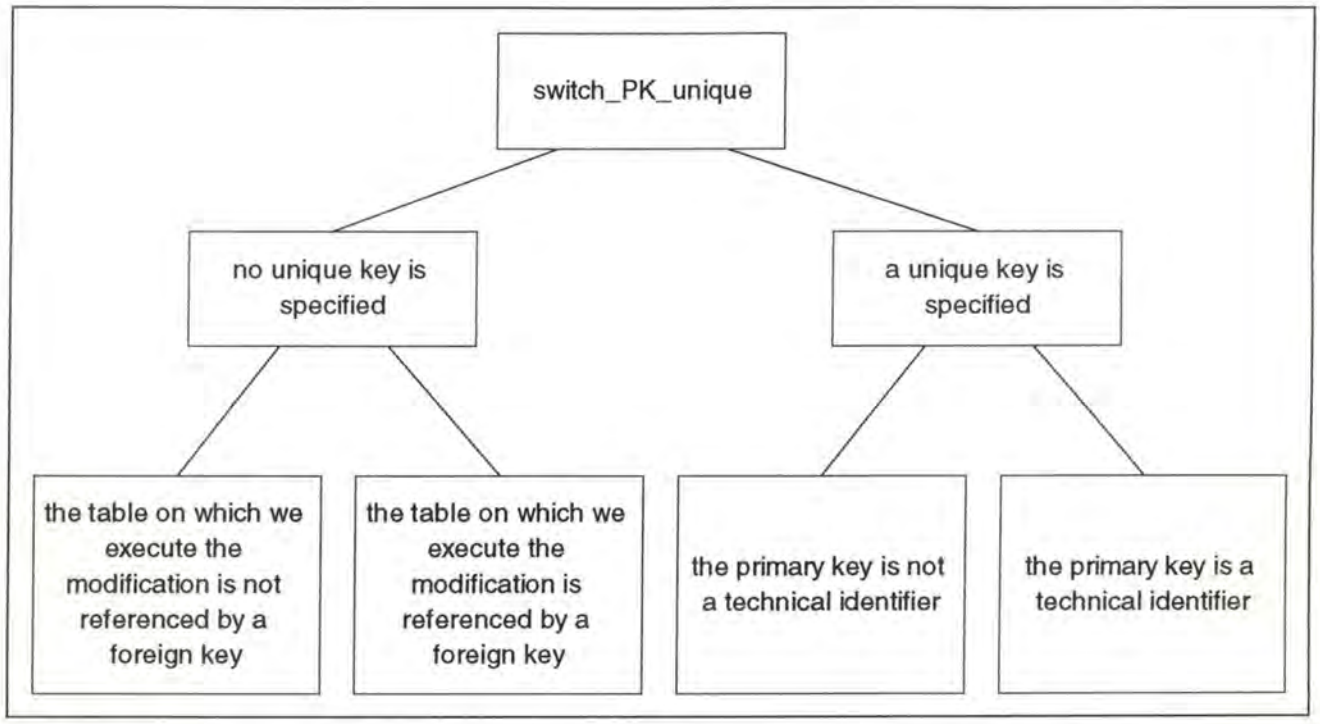


Figure 4 - 30 : Structure of the modification switch_PK_unique

For each one of the four basic cases we will reconsider Figure 4-30 indicating in bold the current position.

3.6.2.1. There is no unique key specified

Let us suppose we have the entity-type ADDRESS and that we want to transform the primary key nadd into a unique key.

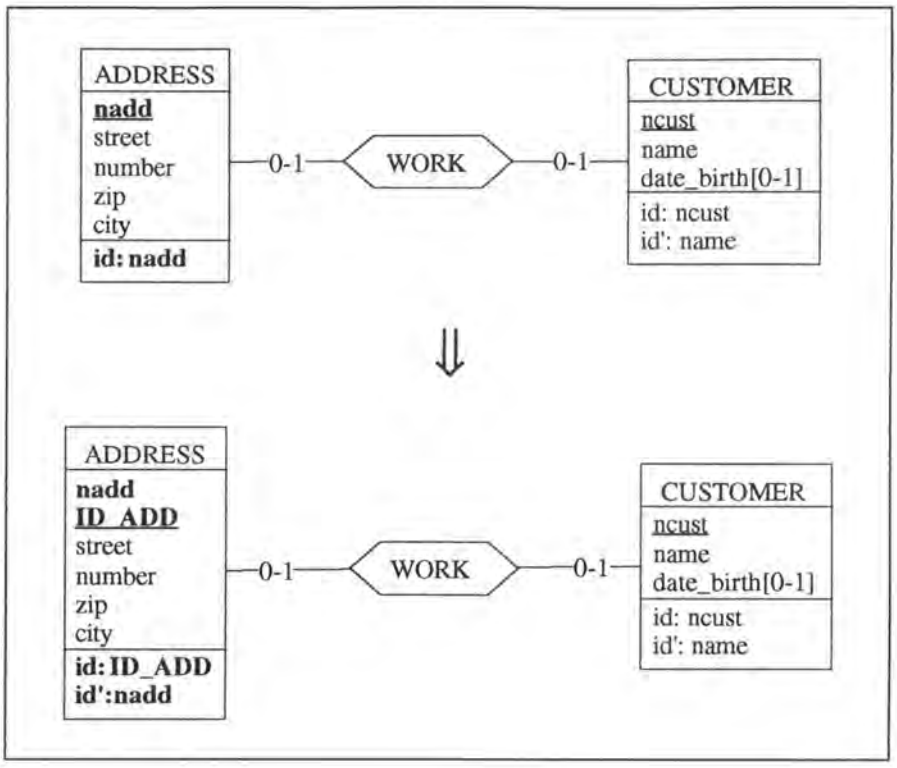
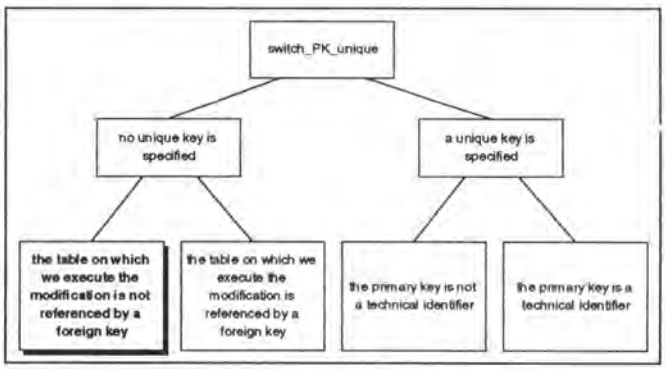


Figure 4 - 31 : Transforming a primary key into a unique key when no unique key is specified, on the conceptual level

Two different cases must be considered:

- WORK is implemented by a foreign key in ADDRESS
- WORK is implemented by a foreign key in CUSTOMER

3.6.2.1.1. WORK is implemented by a foreign key in ADDRESS



3.6.2.1.1.1. Logical Schema

We transform the primary key into a unique key, create a technical identifier and promote it to a primary key:

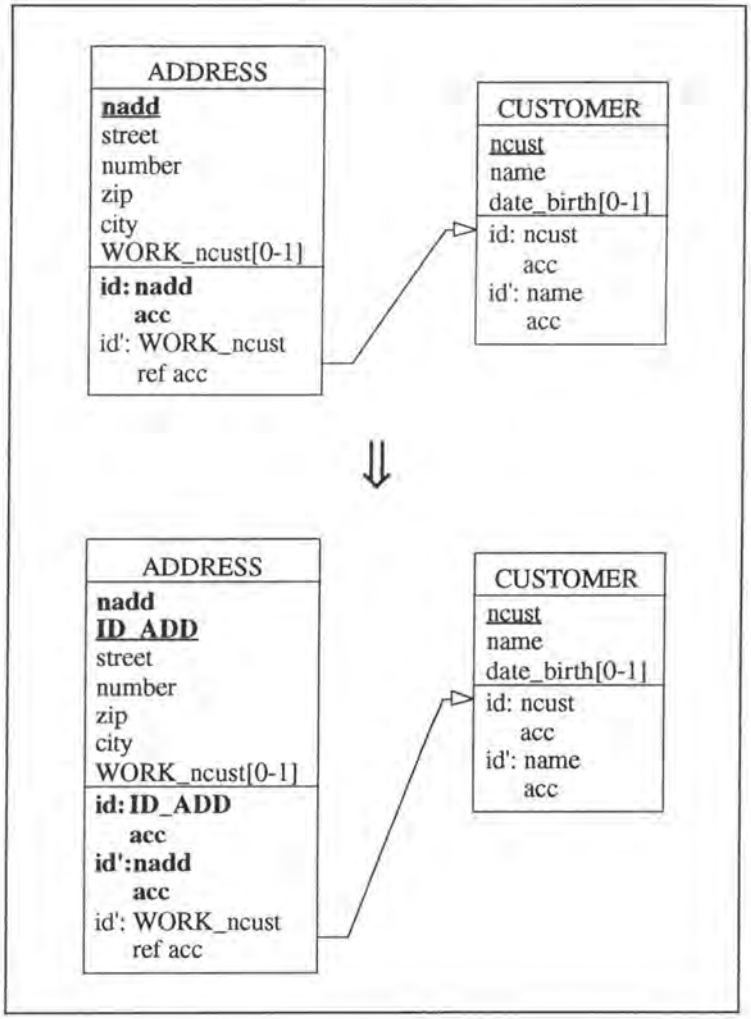


Figure 4 - 32 : Transforming a non referenced primary key into a unique key when no unique key is specified, on the logical level

3.6.2.1.1.2. SQL Description & Data

```
var i: INTEGER;

exec SQL
      (* we create the technical identifier column *)
      alter table ADDRESS
        add ID_ADD smallint  default 0 not null constraint A_ID_ADD;
      (* we assign identifying values to that column *)
      declare c cursor for
        select ID_ADD
          from ADDRESS
        for update of ID_ADD in ADDRESS;
      open c;
      fetch c;
end exec;
i:= 1;
while SQLCODE = 0       (* the last item has not yet been treated *)
do begin
      exec SQL
           update ADDRESS
             set ID_ADD = :i
             where current of c;
           fetch c;
```

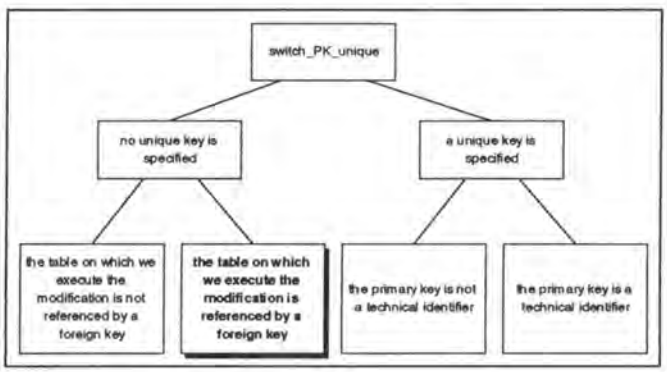
```
      end exec;
      i:= i+1;
   end;
exec SQL
      close c;
      (* we operate the 'real switch' *)
      alter table ADDRESS
        drop constraint idADD1  (* we drop the old primary key
                                   constraint *),
        add constraint primary key(ID_ADD) constraint idADD1,
        add constraint unique(nadd) constraint idADD3;
end exec;
```

No data is lost as we only manipulate identifying features and add a technical identifier.

3.6.2.1.1.3. Program Extracts

There is no impact on the application programs.

3.6.2.1.2. WORK is implemented by a foreign key in CUSTOMER



3.6.2.1.2.1. Logical Schema

We transform the primary key nadd into a unique key, create a technical identifier which we promote to a primary key and change also the foreign key referencing table ADDRESS.

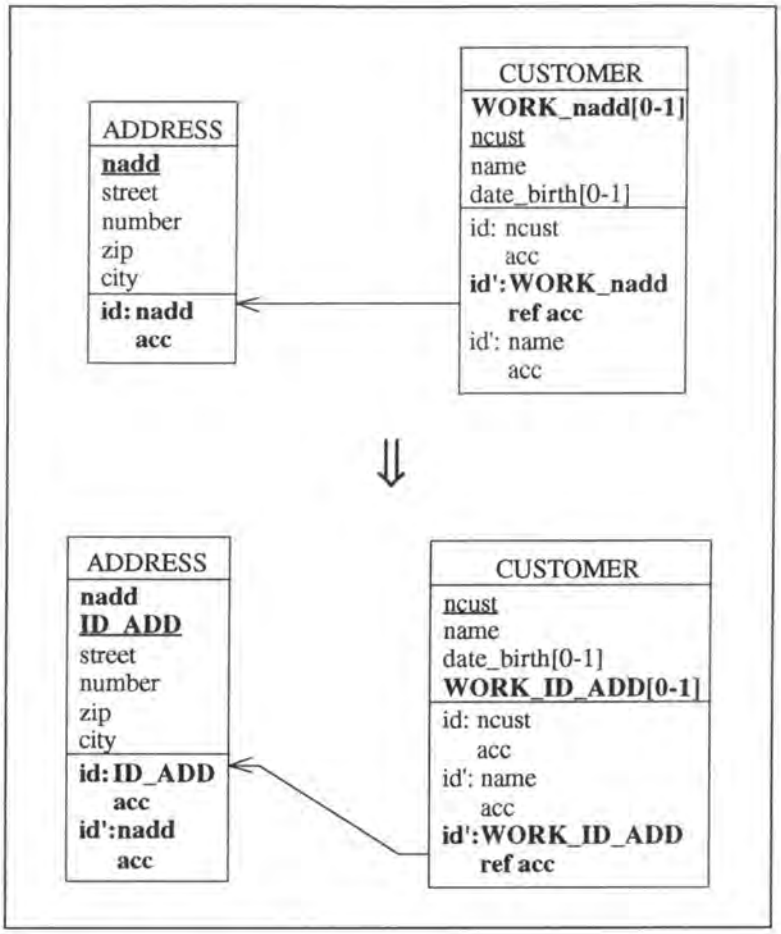


Figure 4 - 33 : Transforming a referenced primary key into a unique key when no unique key is specified, on the logical level

3.6.2.1.2.2. SQL Description & Data

```
var add, i, idADD: INTEGER;

exec SQL
      (* we create the technical identifier column *)
      alter table ADDRESS
        add ID_ADD smallint  default 0 not null constraint A_ID_ADD;
      (* we assign identifying values to that column *)
      declare c1 cursor for
        select ID_ADD
          from ADDRESS
        for update of ID_ADD in ADDRESS;
      open c1;
      fetch c1;
end exec;
i:= 1;
while SQLCODE = 0       (* the last item has not yet been treated *)
do begin
      exec SQL
            update ADDRESS
              set ID_ADD = :i
              where current of c1;
            fetch c1;
      end exec;
      i:= i+1;
   end;
```

```
exec SQL
      close c1;
      (* we replace the foreign key column representing the
         relationship-type WORK *)
      alter table CUSTOMER
        add WORK_ID_ADD smallint,
        drop constraint ADD1;
      declare c2 cursor for
        select WORK_nadd
          from CUSTOMER
          where WORK_nadd is not null
        for update of WORK_ID_ADD;
      open c2;
      fetch c2 into:add
end exec;
while SQLCODE = 0  (* the last item has not yet been treated *)
do begin
      exec SQL
           select ID_ADD
             into :idADD
             from ADDRESS
             where nadd = :add;
           update CUSTOMER
             set WORK_ID_ADD = :idADD
             where current of c2;
           fetch c2 into :add;
      end exec;
   end;
exec SQL
      (* we operate the 'real switch' and adapt the foreign key
         constraints *)
      alter table ADDRESS
        drop constraint idADD1,     (* we drop the old primary key
                                       constraint *)
        add constraint primary key(ID_ADD)constraint idADD1,
        add constraint unique(nadd) constraint idADD2;
      alter table CUSTOMER
        drop constraint idCUS3,
        add constraint foreign key(WORK_ID_ADD) references ADDRESS
                                                  constraint ADD1,
        add constraint unique(WORK_ID_ADD) constraint idCUS3,
        drop WORK_nadd;
      close c2;
end exec;
```

No data is lost as we only manipulate identifying features, add a technical identifier and 'copy' the data representing relationship-type WORK from column WORK_nadd into column WORK_ID_ADD.

3.6.2.1.2.3. Program Extracts

We have to review all the application programs referencing the foreign key representing relationship-type WORK. For example:

-
```
var name: STRING[12];

exec SQL
      select name
        into :name
        from CUSTOMER
        where WORK_nadd = 102;
end exec;
if SQLCODE = 0
then write(name);
```

⇓

```
var name: STRING[12];

exec SQL
      select name
        into :name
        from CUSTOMER
        where WORK_ID_ADD = 52;
            (* Let us suppose ADDRESS has the value 52 for ID_ADD
               if it had the value 102 for nadd *)
end exec;
if SQLCODE = 0
then write(name);
```

-
```
select street, city
  from ADDRESS
  where nadd in ( select WORK_nadd
                    from CUSTOMER
                    where name like '%Dupont%' )
```

⇓

```
select street, city
  from ADDRESS
  where ID_ADD in ( select WORK_ID_ADD
                      from CUSTOMER
                      where name like '%Dupont%' )
```

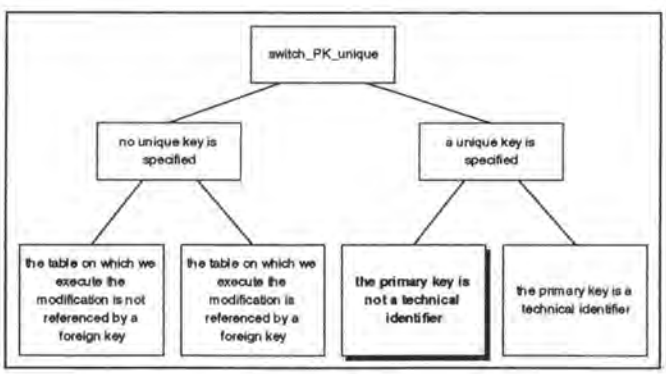
3.6.2.2. The unique key is specified

We have here to distinguish again two cases:

- The primary key is not a technical one
- The primary key is a technical one

For each of these two cases we would have to distinguish again whether the table on which we execute the modification is referenced by a foreign key or not. As these subcases would not bring any new ideas, we will not distinguish them.

3.6.2.2.1. The primary key is not a technical one



Let us suppose we want to replace the primary key ncust of CUSTOMER by the unique key name.

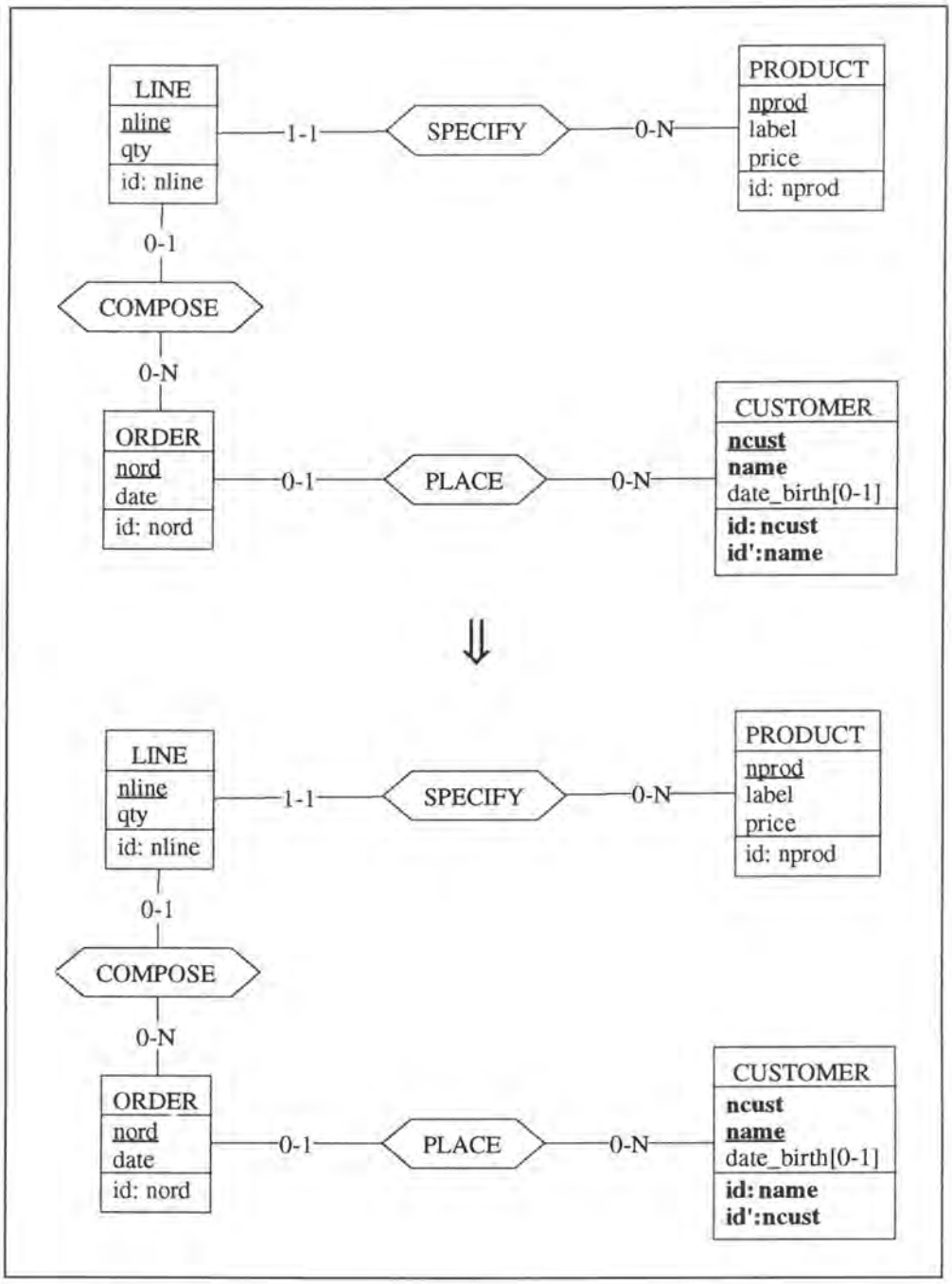


Figure 4 - 34 : Replacing a non technical primary key by a unique key on the conceptual level

3.6.2.2.1.1. Logical Schema

We transform the primary key ncust into a unique key, make the unique key name a primary key and change also the foreign key referencing table ORDER.

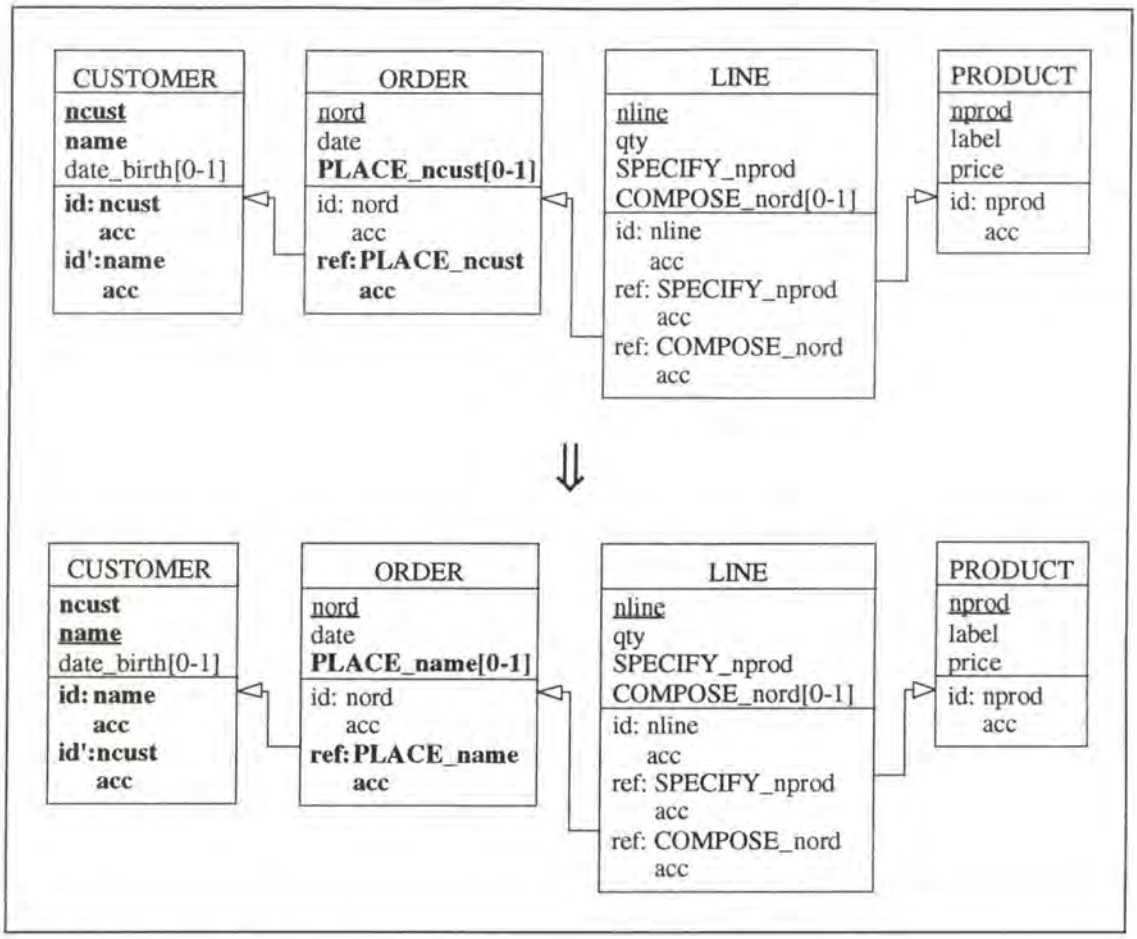


Figure 4 - 35 : Replacing a non technical primary key by a unique key on the logical level

3.6.2.2.1.2. SQL Description & Data

```
var cust: STRING[4];
    name: STRING[12];

exec SQL
      (* we replace the foreign key column representing the
         relationship-type PLACE *)
      alter table ORDER
        add PLACE_name char(12),
        drop constraint CUS1;
      declare c cursor for
        select PLACE_ncust
          from ORDER
          where PLACE_ncust is not null
        for update of PLACE_name;
      open c;
      fetch c into:cust
end exec;
while SQLCODE = 0  (* the last item has not yet been treated *)
do begin
      exec SQL
           select name
             into :name
             from CUSTOMER
             where ncust = :cust;
           update ORDER
             set PLACE_name = :name
```

```
            where current of c;
          fetch c into :cust;
      end exec;
   end;
exec SQL
      (* we operate the 'real switch' and adapt the foreign key
         constraints *)
      alter table CUSTOMER
        drop constraint idCUS1,     (* we drop the old primary key
                                       constraint *)
        drop constraint idCUS2,     (* we drop the old unique key
                                       constraint *)
        add constraint primary key(name) constraint idCUS1,
        add constraint unique(ncust) constraint idCUS2;
      alter table ORDER
        add constraint foreign key(PLACE_name) references CUSTOMER
                                               constraint CUS1,
        drop PLACE_ncust;
      close c;
end exec;
```

No data is lost as we only manipulate identifying features and 'copy' the data representing relationship-type PLACE from column PLACE_ncust into column PLACE_name.

3.6.2.2.1.3. Program Extracts

The second SELECT query of our case study (see page 4-7) must be modified as follows:

```
select *
  from CUSTOMER
  where name in (select PLACE_name
                 from ORDER
                 where nord in (select COMPOSE_nord
                                from LINE
                                where SPECIFY_nprod = 'AA110'))
```

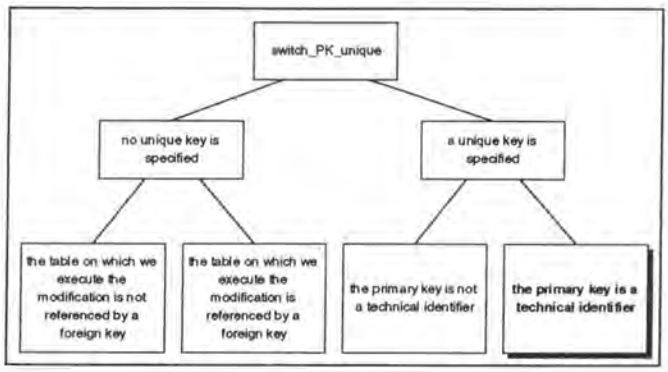
The JOIN query (see page 4-8) becomes:

```
select name, nord
  from CUSTOMER, ORDER
  where (name = PLACE_name) and (date_birth < 01/01/1977).
```

In fact, every program extract referencing PLACE_ncust must be reviewed. For example, let us suppose that the user must give the number of a CUSTOMER (ncust) in order to get his ORDERs. As relationship-type PLACE is now represented by the foreign key PLACE_**name**, either the user has to indicate the name of the CUSTOMER or we have to insert the following query before executing the remaining of the program:

```
select name
  from CUSTOMER
  where ncust = <the number given by the user>
```

3.6.2.2.2. The primary key is a technical one



Let us suppose we have the entity-type ADDRESS where the primary key is a technical identifier and nadd is a unique key. We want now to make nadd a primary key and drop the technical identifier ID_ADD.

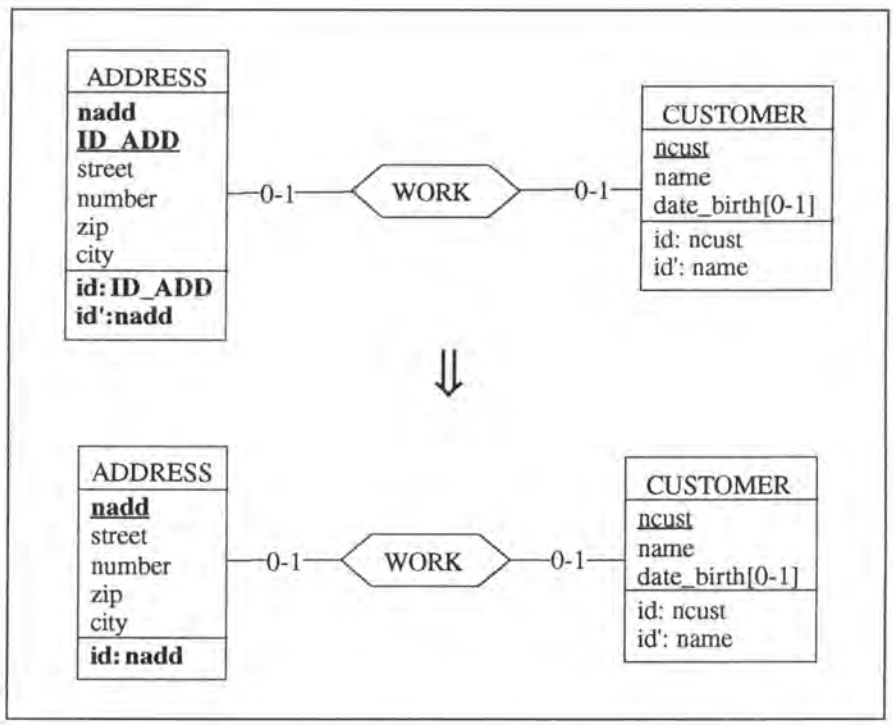


Figure 4 - 36 : Replacing a technical primary key by a unique key on the conceptual level

3.6.2.2.2.1. Logical Schema

In order to simplify, we only consider the case where the relationship-type WORK has been implemented by the foreign key in relation ADDRESS. We have to make nadd a primary key and drop ID_ADD.

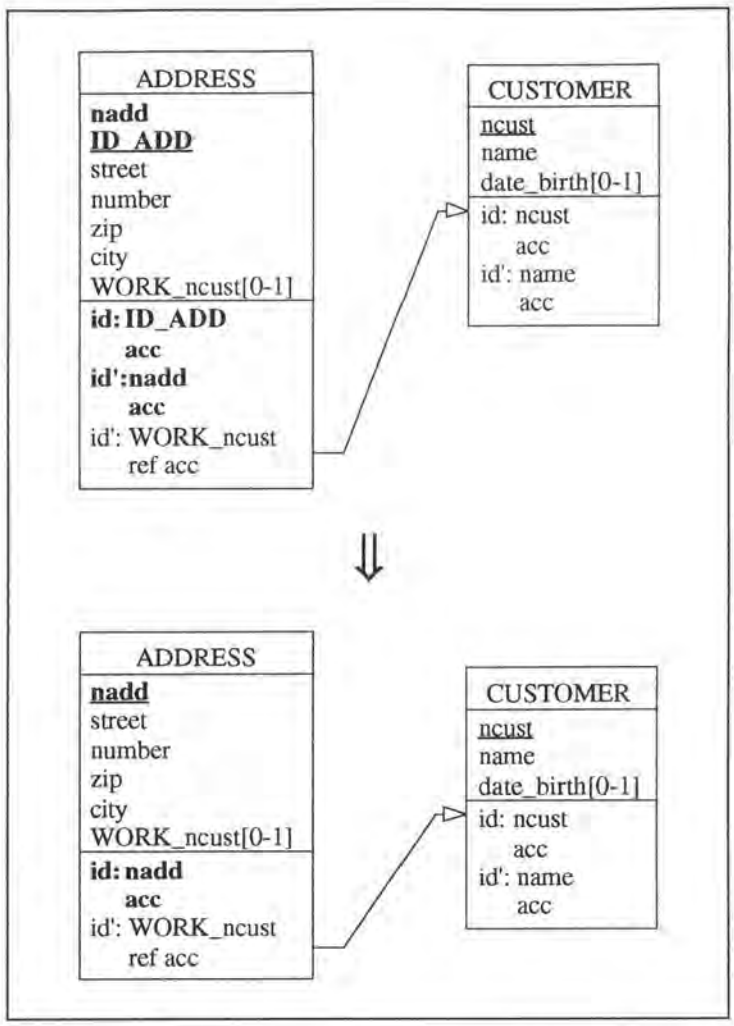


Figure 4 - 37 : Replacing a technical primary key by a unique key on the logical level

3.6.2.2.2. SQL Description & Data

```
alter table ADDRESS
  drop constraint idADD1,      (* we drop the old primary key
                                  constraint *)
  drop constraint idADD2,      (* we drop the old unique key
                                  constraint *)
  drop constraint A_ID_ADD,    (* we remove the mandatory feature from
                                  column ID_ADD *)
  add constraint primary key(nadd) constraint idADD1,
  drop ID_ADD;
```

No data is lost as we do not consider the information included in column ID_ADD as semantical data.

3.6.2.2.3. Program Extracts

There is no impact on the application programs as ID_ADD is a technical construct and is thus not referenced by any query.

Chapter 5:

Study of the Modifications: General Approach

1. INTRODUCTION

After having studied some modifications on a case study, we will analyse the same modifications in general. As this chapter is a very technical one and is also partly redundant with the previous one, it can be skipped in a first reading of our thesis.

Let us repeat that we have here again to study the modifications of the conceptual level and their impact on the logical level, on the SQL database structure, on the data and on the application programs. This is illustrated by Figure 5-1.

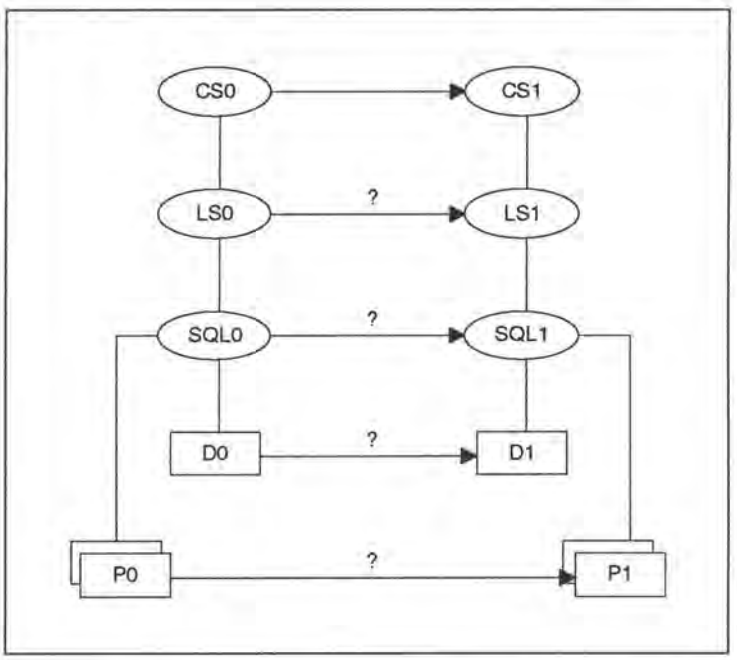


Figure 5 - 1 : Representation of the database evolution problem

If the conceptual schema CS0 has been changed, the logical schema LS0 and the SQL description SQL0 must be changed accordingly. Data D0 is no longer valid and has to be converted into data D1. Finally the applications P0 must be partly rewritten in order to comply with the new data structures described in SQL1.[HAI94a]

As shown in the third chapter, the modifications are classified according to the objects on which they apply on one hand and, on the other hand, according to their nature: augmenting, decreasing or preserving semantics (see page 3-4).

As we will not describe all the modifications, we will give once more the typology of the modifications, indicating in bold those that we will analyse in detail in this chapter. The modifications that will not be treated here can be found in appendix 2.

Modifications of the entity-types:	add_entity-type remove_entity-type **rename_entity-type**
Modifications of the relationship-types:	add_1-1/0-1_rel-type add_0-1/0-1_rel-type add_1-1/0-N_rel-type add_0-1/0-N_rel-type remove_1-1/0-1_rel-type **remove_0-1/0-1_rel-type** remove_1-1/0-N_rel-type remove_0-1/0-N_rel-type rename_1-1/0-1_rel-type rename_0-1/0-1_rel-type rename_1-1/0-N_rel-type rename_0-1/0-N_rel-type
Modifications of the roles:	**augment_max_card** decrease_min_card decrease_max_card augment_min_card
Modifications of the attributes:	add_optional_attribute add_mandatory_attribute make_attr_optional extend_domain_attribute change_type_int_char change_type_float_char change_type_date_char change_type_date_int change_type_int_float change_type_date_float remove_optional_attribute remove_mandatory_attribute **make_attr_mandatory** restrict_domain_attribute change_type_char_int change_type_float_int change_type_char_float change_type_char_date change_type_int_date change_type_float_date rename_optional_attribute rename_mandatory_attribute
Modifications of the identifiers:	remove_unique_feature add_unique_feature **switch_PK_unique**

For each object, we will study the same modifications as those described in chapter 4, here however in general. Each modification is here also decomposed into two parts: its classification and its description. The description is again decomposed into three subparts: the impact of the modification on the Logical Schema, on the SQL Description & Data and on the Program Extracts.

2. STUDY OF THE MODIFICATIONS: GENERAL APPROACH

2.1. RENAME_ENTITY-TYPE[1]

2.1.1. Classification of the Modification

As shown in Figure 5-2, rename_entity-type is a modification on entity-types which preserves the semantics.

semantics → / objects ↓	augmenting	decreasing	preserving
entity-type			X
rel-type			
role			
attribute			
identifier			

Figure 5 - 2 : Classification of rename_entity-type

2.1.2. Description of the Modification

Let us suppose we want to rename the entity-type E into E1.

[1]Normally we would have to add here the following precondition: 'The new name of the entity-type that should be renamed must not yet exist.' As such preconditions are trivial, we will not indicate them anymore.

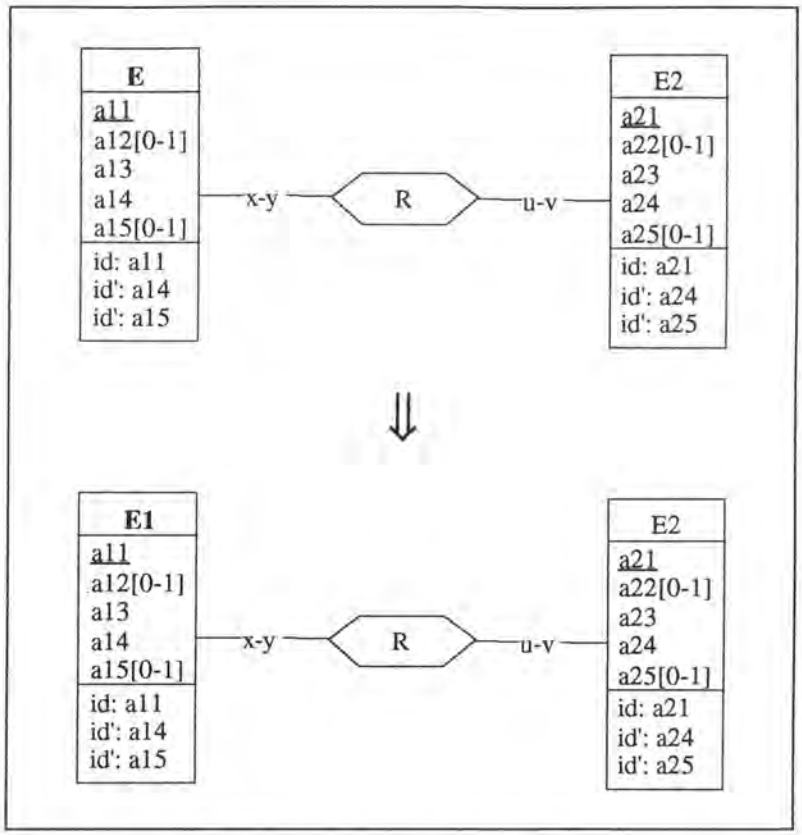


Figure 5 - 3: Renaming an entity-type on the conceptual level

2.1.2.1. Logical Schema

In the logical schema, we have to change the name of the corresponding relation. Due to the parametrical cardinalities, different cases are possible. In Figure 5-4 however (and only there), we will only illustrate the two basic ones:

- R is represented by a foreign key in E
- R is represented by a foreign key in E2

For each of these two cases, we will only consider the situation where relationship-type R has one 0-N role. The other cases would be similar, except that we would have to express identifying features.

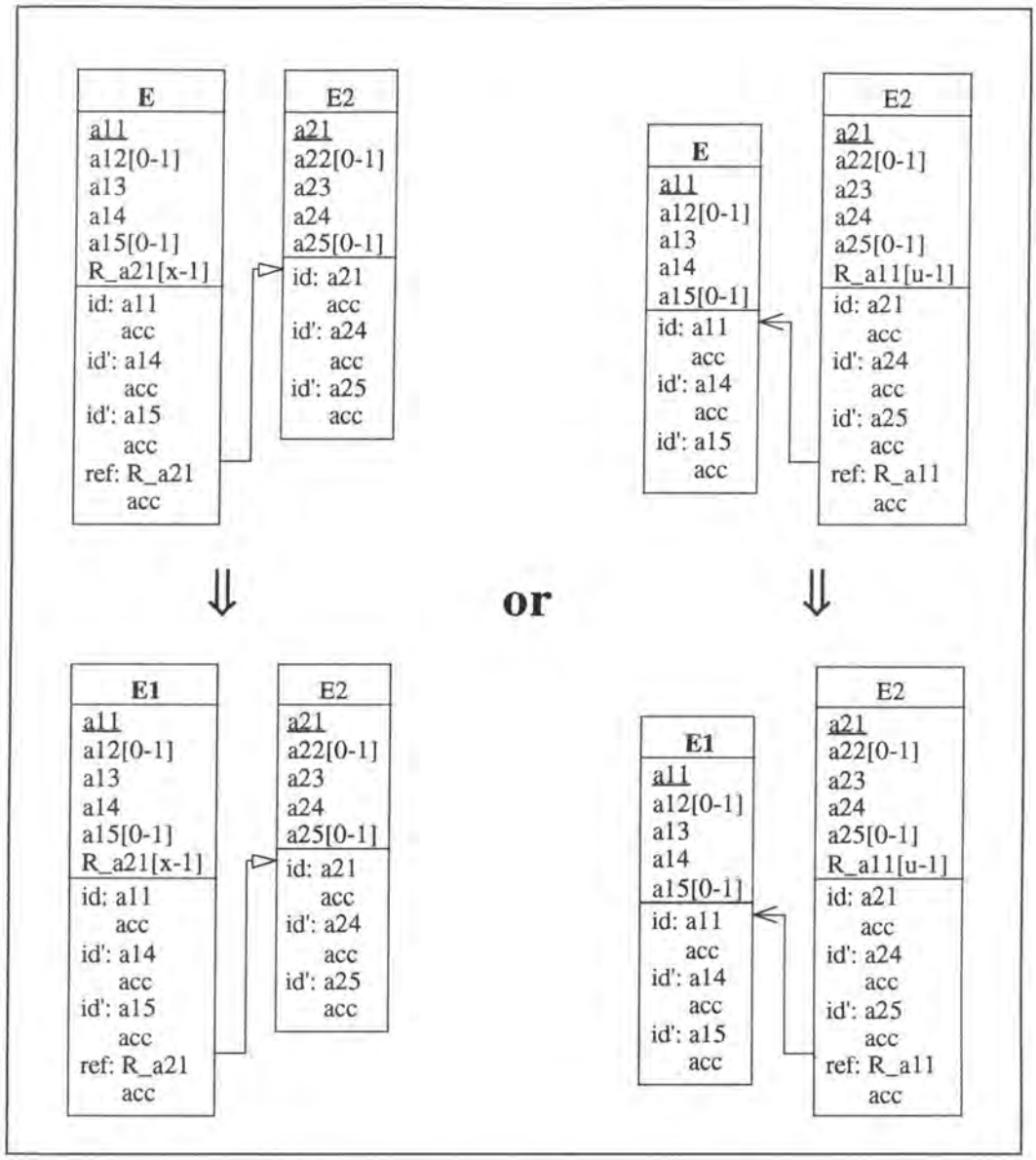


Figure 5 - 4 : Renaming an entity-type on the logical level

2.1.2.2. SQL Description & Data

In some SQL languages there may be a 'rename table' command. The modification would then become:

```
alter table E
  rename table E1 on cascade;
```

In SQL-RDB however, no such command exists and we have therefore to create a new table and to copy the data into it.

```
exec SQL
      (* we create table E1 *)
      create table E1
            ( a11 <type>  not null  constraint E1_a11,
```

```
                a12 <type>,
                a13 <type>  not null  constraint E1_a13,
                a14 <type>  not null  constraint E1_a14,
                a15 <type>,
                primary key (a11) constraint idE1_#[2],
                unique (a14) constraint idE1_#,
                unique (a15) constraint idE1_# )
end exec;
(* we create the foreign keys in E1 *)
for each of the relationship-types R connected to E
do if R is represented by a foreign key in E
   then begin
             if x = 0
             then exec SQL
                       alter table E1
                         add R_a21  <type>;
                  end exec
             else exec SQL
                       alter table E1
                         add R_a21  <type>  default <value>  not null
                                                  constraint E1_R_a21;
                  end exec;
             if v = 1
             then exec SQL
                       alter table E1
                         add constraint unique (R_a21)  constraint idE1_#;
                  end exec;
             exec SQL
                  alter table E1
                    add constraint foreign key (R_a21) references E2
                                                  constraint E2_#;
             end exec;
          end;
exec SQL
      (* we insert the data of E into E1 *)
      insert into E1
        select *
        from E
end exec;
(* we redirect to E1 the foreign keys referencing E *)
for each of the relationship-types R connected to E
do if R is represented by a foreign key in E2
   then exec SQL
                alter table E2
                  drop constraint E_#;  (* we remove the old foreign key
                                           feature *)
                  add constraint foreign key (R_a11) references E1
                                                  constraint E1_#,
          end exec;

(* For each view defined on table E, we have to redefine it on E1. In
future we will not consider views anymore as they do not correspond to ER
objects. *)

drop table E cascade;
```

No data is lost as the data is just moved from one table into another.

Notes:

- This operation in SQL-RDB is often a very slow one as we have to copy a whole table. We thus recommend to create a view E1 which includes only the table E. This could be realized by the following command:

[2]As it is difficult to indicate the proper number for each constraint, we will use the symbol #.

```
create view E1
  as select *
       from E
```

- Other SQL languages, such as DB2, offer another possibility to implement the modification: giving a synonym to the entity-type (that has to be renamed) instead of renaming it properly. This alternative could be realised by the following SQL command:

```
create synonym E1
  for E
```

Note that in both cases the original table is however not renamed.

2.1.2.3. Program Extracts

- In all the select queries referencing E, we have to rename it with E1.
  For example:

```
select ...
  from E
  where ...
```

```
select ...
  from E1
  where ...
```

- In the following example, we have to rename E not only in the 'from' clause, but also in the 'where' clause:

```
select ...
  from E, E2
  where E.a = E2.a
```

```
select ...
  from E1, E2
  where E1.a = E2.a
```

- In addition to the select queries, we have also to review the application programs in which they appear. For instance, we must rename certain variables and/or some fields or headings in the user interfaces. Finally, let us note that the documentation should also be updated.

2.2. REMOVE_0-1/0-1_REL-TYPE

2.2.1. Classification of the Modification

As shown in Figure 5-5, remove_0-1/0-1_rel-type is a modification on relationship-types which decreases the semantics.

semantics → / objects ↓	augmenting	decreasing	preserving
entity-type			
rel-type		X	
role			
attribute			
identifier			

Figure 5 - 5 : Classification of remove_0-1/0-1_rel-type

2.2.2. Description of the Modification

Let us suppose that we want to remove the 0-1/0-1 relationship-type R between the entity-types E1 and E2.

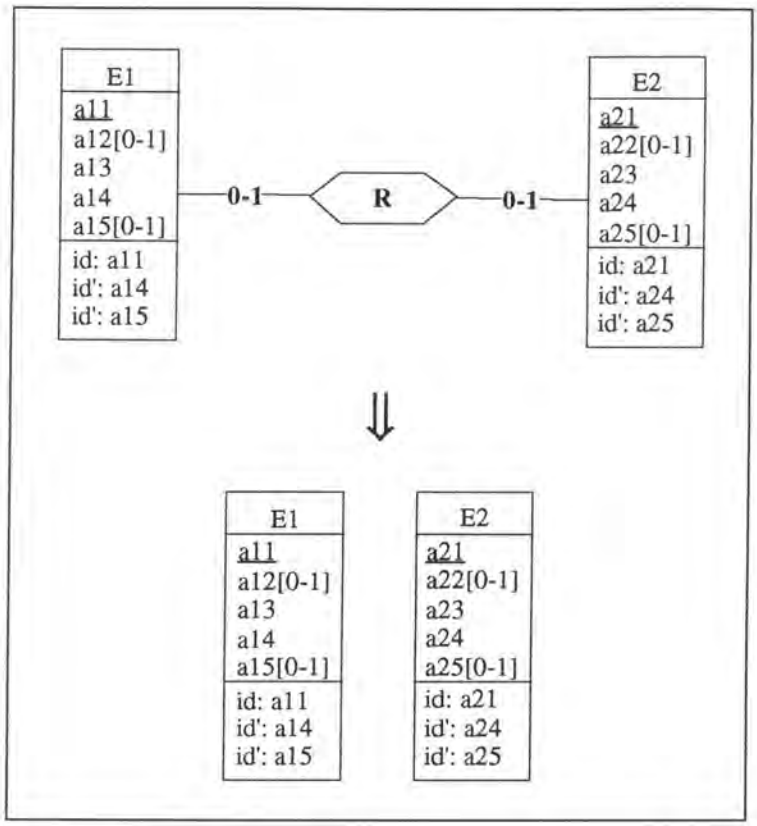


Figure 5 - 6 : Removing a 0-1/0-1 relationship-type on the conceptual level

2.2.2.1. Logical Schema

Depending on the way R has been implemented, we remove either column R_a21 from E1 or column R_a11 from E2 with its candidate and foreign key features.

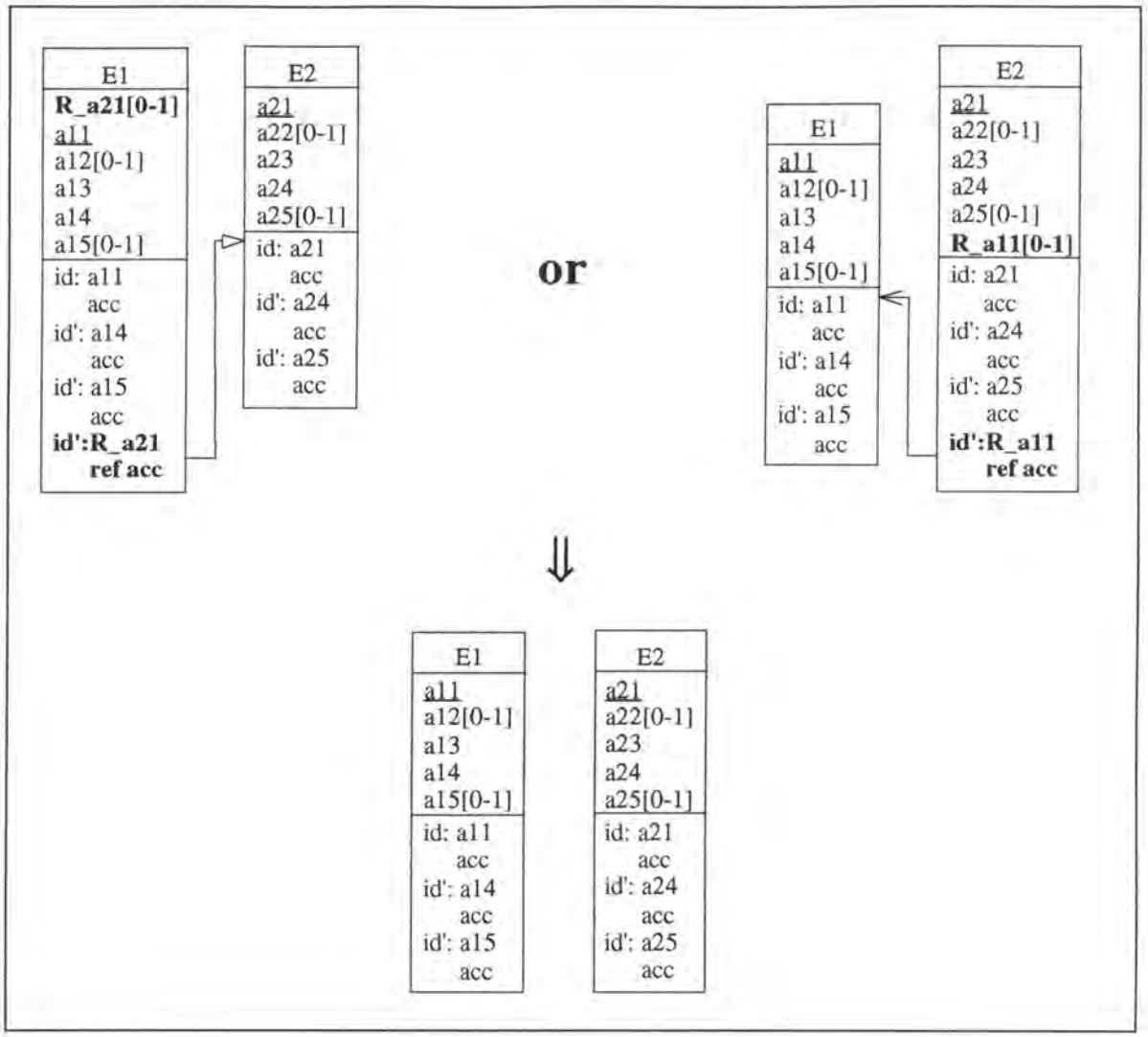


Figure 5 - 7 : Removing a 0-1/0-1 relationship-type on the logical level

2.2.2.2. SQL Description & Data

```
if R is implemented by a foreign key in E1
then exec SQL
            alter table E1
              drop constraint idE1_#,      (* we remove the unique key
                                              feature *)
              drop constraint E2_#,        (* we remove the foreign key
                                              feature *)
              drop R_a21;
      end exec
else   (* R is implemented by a foreign key in E2 *)
      exec SQL
            alter table E2
              drop constraint idE2_#,      (* we remove the unique key
                                              feature *)
              drop constraint E1_#,        (* we remove the foreign key
                                              feature *)
              drop R_a11;
      end exec;
```

The link, representing R, between tables E1 and E2 is lost.

2.2.2.3. Program Extracts

Application programs in which select queries referencing R_a21 in E1 (or R_a11 in E2) appear must be reviewed. We cannot describe a general method how to deal with these application programs as each one of them must be treated individually, depending on its context. A CASE tool offering this modification should indicate the concerned program extracts and should sometimes give hints about the way how to change them. The user has then to check whether the variables are still all needed and he has also to update the documentation. Finally, he must change certain user interfaces (for an example see page 4-16).

2.3. AUGMENT_MAX_CARD

2.3.1. Classification of the Modification

As shown in Figure 5-8, augment_max_card is a modification on roles which augments the semantics.

semantics → / objects ↓	augmenting	decreasing	preserving
entity-type			
rel-type			
role	X		
attribute			
identifier			

Figure 5 - 8 : Classification of augment_max_card

2.3.2. Description of the Modification

Precondition:
Given the restrictions of the relationship-types in the Kernel (see page 3-2), the only augmentations of the maximum cardinality of a role that we accept so far are:

- 1-1/0-1 → 1-1/0-N
- 0-1/0-1 → 0-1/0-N

We want to augment to N the maximum cardinality of the 0-1 role of relationship-type R.

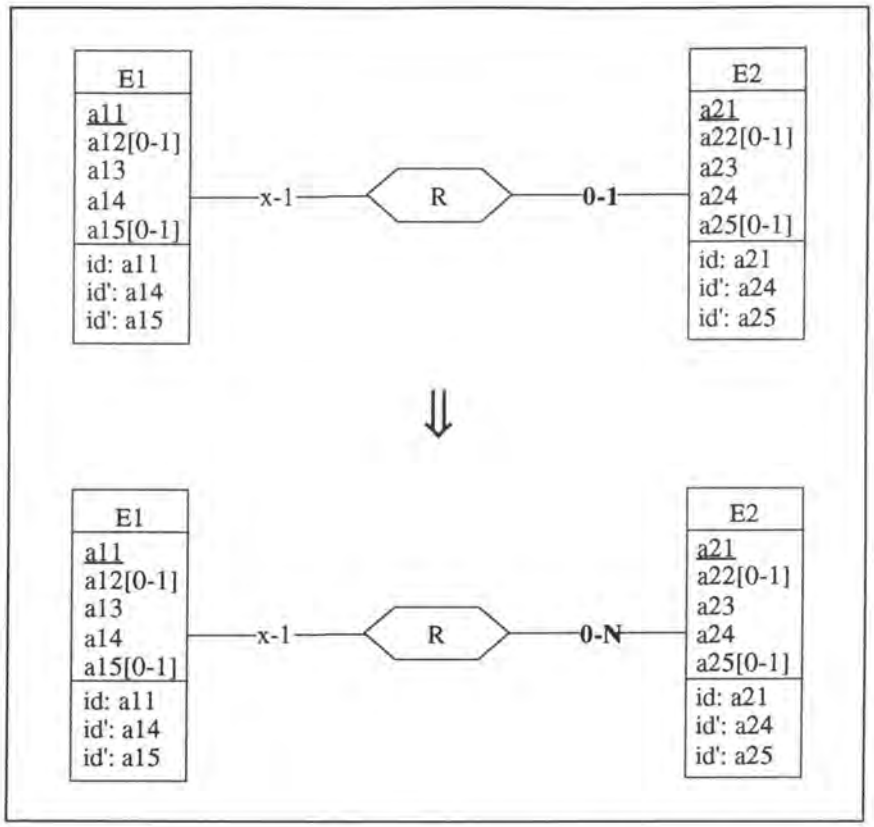


Figure 5 - 9 : Augmenting the maximum cardinality of a role to N on the conceptual level

2.3.2.1. Logical Schema

We have either to remove the candidate key from R_a21 in E1 or to replace the foreign key R_a11 in E2 by a (non unique) foreign key R_a21 in E1.

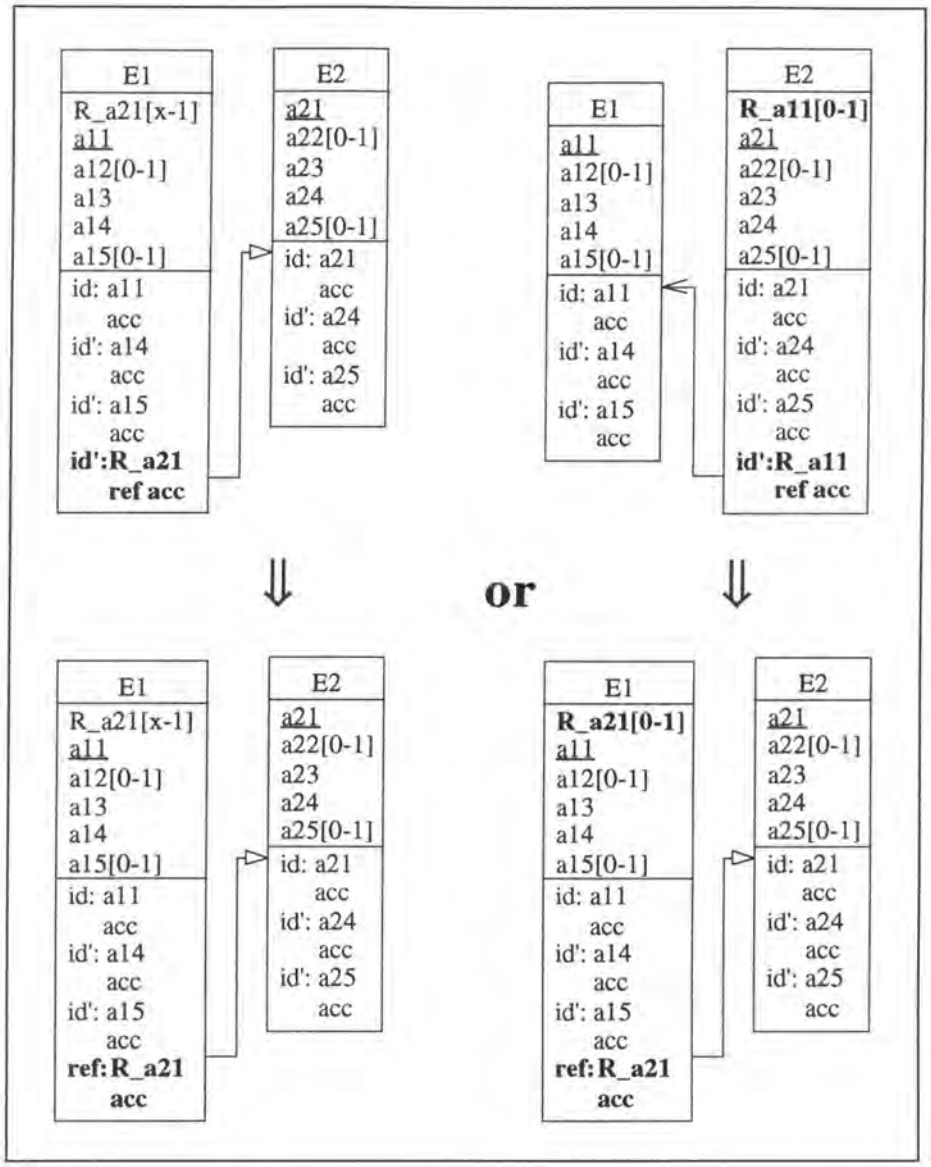


Figure 5 - 10 : Augmenting the maximum cardinality of a role to N on the logical level

2.3.2.2. SQL Description & Data

```
var a21: <type>;
    a11: <type>;

if the foreign key representing R is in E1
then exec SQL
           alter table E1
             drop constraint idE1_#;
     end exec      .
else (* the foreign key representing R is in E2 *)
      begin
        exec SQL
            (* we create the new foreign key column *)
            alter table E1
              add R_a21  <type>;
```

```
        (* we copy the data representing the relationship-type R from
           table E2 into table E1 *)
        declare c cursor for
          select a21, R_a11
            from E2
            where R_a11 is not null;
        open c;
        fetch c into :a21, :a11;
    end exec;
    while SQLCODE = 0      (* the last item has not yet been treated *)
    do begin
          exec SQL
              update E1
                set R_a21 = :a21
                where a11 = :a11;
              fetch c into :a21, :a11;
          end exec;
       end;
    exec SQL
        (* we add and remove the necessary constraints *)
        alter table E1
          add constraint foreign key (R_a21) references E2
                                                  constraint E2_#;
        alter table E2
          drop constraint idE2_#,
          drop constraint E1_#,
          drop R_a11;
        close c;
    end exec;
end;
```

Note that no data is lost as either no changes are applied on the data or the data is only 'copied' from relation E2 into relation E1.

2.3.2.3. Program Extracts

Before considering the select queries, let us note that the user has to replace certain variables by arrays, that he has to review certain user interfaces and that he has also to update the documentation. In order to study the impact of the modification on the select queries, we must distinguish whether the foreign key representing R was in E1 or E2.

2.3.2.3.1. The foreign key representing R was in E1

As the foreign key R_a21 is not identifier of E1 anymore, several rows can now have the same value for column R_a21. We thus have to define a cursor for 'select ...into ...' queries referencing 'R_a21 = ' in their 'where' clause.

```
var ...

:
exec SQL
      select ...
        into ...
        from E1
        where R_a21 = ...
end exec;
if SQLCODE = 0          (* if such a row has been found *)
then ...
:
```

⇓

```
var ...

   :
exec SQL
      declare c cursor for
        select ...
          from E1
          where R_a21 = ...;
       open c;
       fetch c into ...;
end exec;
while SQLCODE = 0        (* the last item has not yet been treated *)
  do begin
        :
        exec SQL
            fetch c into ...;
        end exec
     end;
exec SQL
      close c
end exec;
   :
```

2.3.2.3.2. The foreign key representing R was in E2

- A similar problem concerning the 'select ...into ...' queries occurs in this case. The select query must here however also be modified.

```
var a11: <type>;

  :
exec SQL
      select R_a11
        into :a11
        from E2
        where a24 = ...;
end exec;
if SQLCODE = 0
then ...
  :
```

```
var a11: <type>;

  :
exec SQL
      declare c cursor for
        select a11
          from E1
          where R_a21 in (select a21
                                  from E2
                                  where a24 = ...);
       open c;
       fetch c into :a11;
end exec;
while SQLCODE = 0        (* the last item has not yet been treated *)
do begin
        :
       exec SQL
            fetch c into :a11;
       end exec;
   end;
exec SQL
       close c
```

```
end exec;
  :
```

- As R is now represented by a foreign key in E1, the select queries referencing R_a11 must be reviewed.

```
- select ...
     from E1
     where a11 in ( select R_a11
                    from E2
                    where ... )
```

```
  select ...
     from E1
     where R_a21 in ( select a21
                      from E2
                      where ... )

- select ...
    from E2
    where R_a11 ...
```

```
  select ...
    from E2
    where a21 in ( select R_a21
                   from E1
                   where a11 ... )
```

2.4. MAKE_ATTR_MANDATORY

2.4.1. Classification of the Modification

As shown in Figure 5-11, make_attr_mandatory is a modification on attributes which decreases the semantics.

semantics → / objects ↓	augmenting	decreasing	preserving
entity-type			
rel-type			
role			
attribute		X	
identifier			

Figure 5 - 11 : Classification of make_attr_mandatory

2.4.2. Description of the Modification

We have to distinguish whether the attribute which we want to make mandatory is a unique key or not. Let us suppose we want to make attribute a12 in entity-type E1 mandatory.

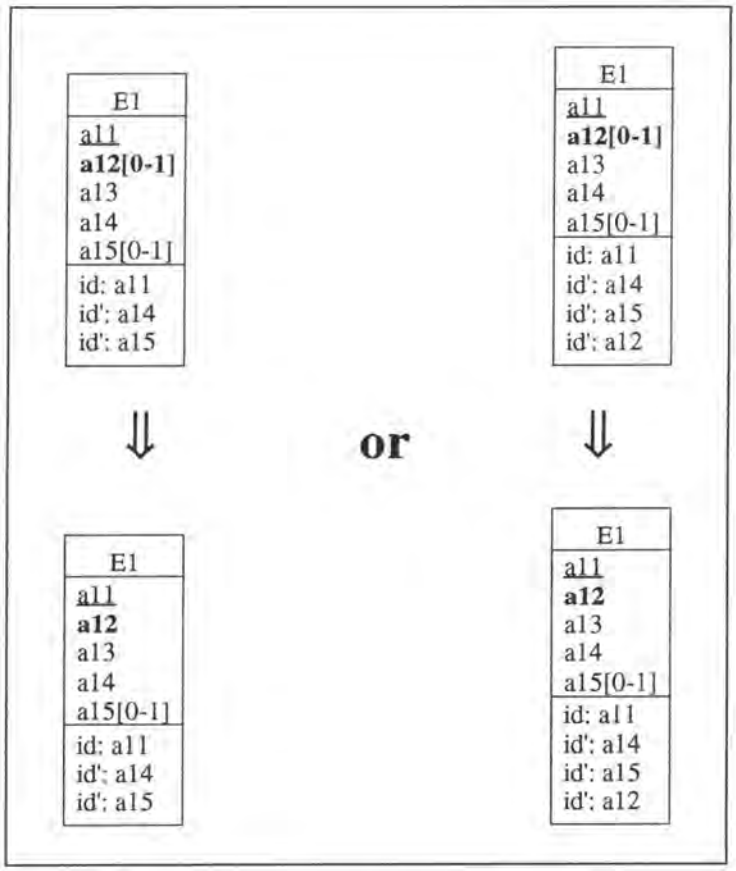


Figure 5 - 12 : Making an attribute mandatory on the conceptual level

2.4.2.1. Logical Schema

We make the column a12 in relation E1 mandatory.

2.4.2.2. SQL Description & Data

```
procedure Delete_on_cascade(r, E);

(* Before deleting a row r in table E, we must delete the rows r1
   'referencing r' or set to null in table E1 the foreign key column of the
   rows r1 'referencing r'. If the rows r1 are deleted, the problem must be
   treated recursively. *)

begin
  for each table E1
  do for each foreign key referencing table E
     do for each of the rows r1 having as foreign column value the value of
           the primary key column of row r
        do if the user wants to avoid the loss of data
           then if the foreign key column (FK) is optional
```

```
                then exec SQL
                        update E1
                          set FK = null
                     end exec
                else call Delete_on_cascade(r1, E1)
           else call Delete_on_cascade(r1, E1);
  exec SQL  (* delete r from E *)
      delete
        from E
        where id = r.id
  end exec;
end;


if a12 is not a unique key
then begin
       if the user wants to avoid the loss of data wherever it is possible
       then exec SQL
                  update E1
                    set a12 = <value>
                    where a12 is null
            end exec
       else for all the rows r of E1 having a null value for column a12
            do call Delete_on_cascade(r, E1);
       exec SQL
            alter table E1
              alter a12 not null constraint E1_a12;
       end exec;
     end
else begin
        (* we cannot use here a default value because of the unique key
           feature of column a12 *)
        for all the rows r of E1 having a null value for column a12
        do call Delete_on_cascade(r, E1);
        exec SQL
            alter table E1    (* we can only modify a column on which no
                                 constraints apply *)
              drop constraint idE1_#,     (* we remove the old unique key
                                             feature *)
              alter a12 not null constraint E1_a12,
              add constraint unique (a12) constraint idE1_#;
        end exec;
      end;
```

It depends on the choice of the user and on the uniqueness feature of the column a12 whether we loose data or not.

2.4.2.3. Program Extracts

Select queries testing the null value of the attribute that has to be made mandatory must be modified or deleted depending on the case.

```
select ...
  from E1
  where a12 is null
```

```
select ...
  from E1
  where a12 = <value>
```

or

(* The user did not want to loose data and a12 is not a unique key *)

(* The user accepted to loose data or a12 is a unique key *)

It is often not sufficient to change or delete the select queries only, we must also review the program extracts in which they appear. For example: in certain cases we do not need the null indicator anymore and certain tests, checking the null value of column a12, must either be changed or dropped.

```
var a12: <type>;
      :
    null_indicator: INTEGER;

exec SQL
      select a12
        into :a12:null_indicator, ...
        from E1
        where a11 = ...
end exec;
if SQLCODE = 0     (* if such a row has been found *)
then if null_indicator = 0
     then ...
```

```
var a12: <type>;
      :

exec SQL
      select a12
        into :a12, ...
        from E1
        where a11 = ...
end exec;
if SQLCODE = 0     (* if such a row has been found *)
then ...
```

2.5. SWITCH_PK_UNIQUE

2.5.1. Classification of the Modification

As shown in Figure 5-13, switch_PK_unique is a modification on identifiers which preserves the semantics.

semantics → / objects ↓	augmenting	decreasing	preserving
entity-type			
rel-type			
role			
attribute			
identifier			X

Figure 5 - 13 : Classification of switch_PK_unique

2.5.2. Description of the Modification

We want to transform the existing primary key into a unique key in entity-type E1 and vice versa. The user has the choice whether to specify a unique key or not. If he does not specify any unique key, then a technical identifier is created as primary key.

Precondition:
If a unique key is specified then it must not be optional as SQL-RDB does not allow optional attributes as primary key.

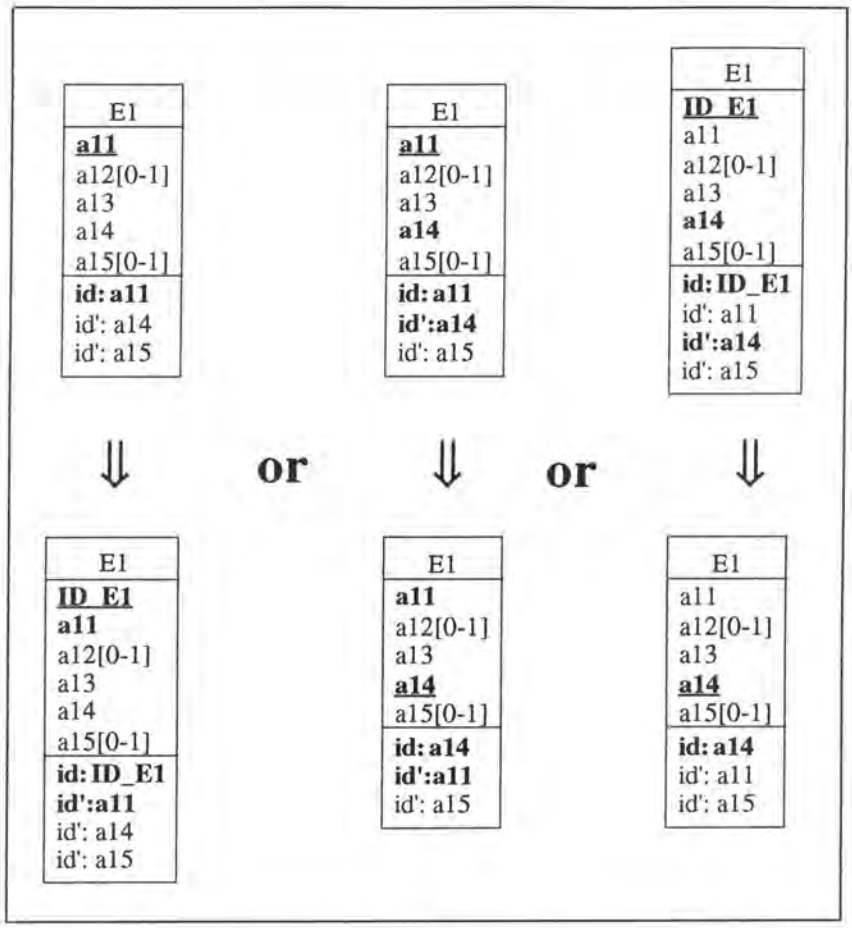


Figure 5 - 14 : Switching the primary key and the unique key on the conceptual level

2.5.2.1. Logical Schema

We transform the existing primary key into a unique key in relation E1, drop it if it was a technical one, create a technical primary key if no unique key was specified and replace the foreign keys referencing relation E1 accordingly.

2.5.2.2. SQL Description & Data

```
var i: INTEGER;
    idADD: INTEGER;

procedure Switch(E1, old_prim, new_prim)

(* This procedure transforms the existing primary key old_prim into a
   unique key in table E1 and the unique key new_prim into the new primary
   key of table E1. *)
```

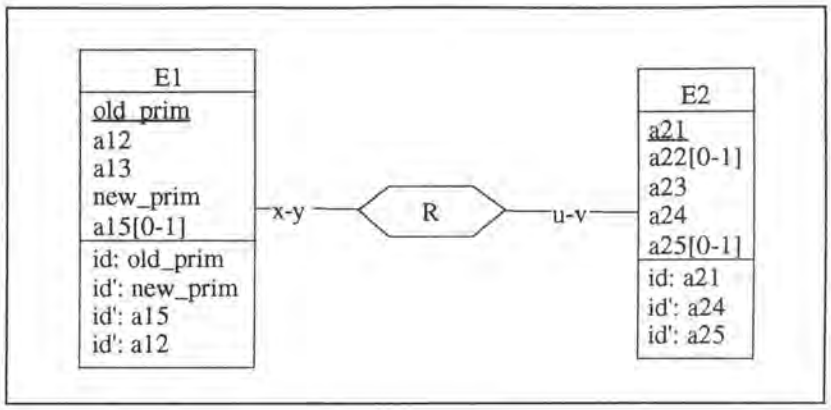


Figure 5 - 15 : General situation used in procedure Switch

```
var old_prim: <type>;
    new_prim: <type>;

begin
      for each foreign key in table E referencing table E1
          and representing relationship-type R
      do begin
            (* we create the new foreign key column, we remove the old
               foreign key constraint and we copy the data representing
               relationship-typ R *)
            if u = 0
            then exec SQL
                        alter table E
                          add R_new_prim <type>,
                          drop constraint E1_#;
                        declare c cursor for
                          select R_old_prim
                            from E
                            where R_old_prim is not null
                          for update of R_new_prim;
                 end exec
            else (* u = 1 *)
                 exec SQL
                        alter table E
                          add R_new_prim <type> default <value>  not null
                              constraint  E_R_new_prim,
                          drop constraint E1_#;
                        declare c cursor for
                          select R_old_prim
                            from E
                          for update of R_new_prim;
                 end exec;
            exec SQL
                 open c;
                 fetch c into :old_prim;
            end exec;
            while SQLCODE = 0 (* the last item has not yet been treated *)
            do exec SQL
                 select new_prim
                   into :new_prim
                   from E1
                   where old_prim = :old_prim;
                 update E
                   set R_new_prim = :new_prim
                   where current of c;
                 fetch c into :old_prim;
               end exec;
            exec SQL
                 close c
```

```
            end exec;
         end;
      if old_prim in E1 is a technical identifier
      then exec SQL
                (* we drop the old primary key with its constraints and
                   we add the primary key feature to column new_prim *)
                alter table E1
                  drop constraint idE1_#,
                                  (* primary key feature of old_prim *)
                  drop constraint E1_ID_E1,
                  drop ID_E1,
                  drop constraint idE1_#,
                                  (* uniqueness feature of new_prim *)
                  add constraint primary key (new_prim)
                                                  constraint idE1_#;
            end exec
      else (* old_prim in E1 is not a technical identifier *)
            exec SQL
                  (* We switch the identifying features between new_prim
                     and old_prim *)
                alter table E1
                  drop constraint idE1_#,
                                  (* primary key feature of old_prim *)
                  add constraint unique (old_prim) constraint idE1_#,
                  drop constraint idE1_#,
                                  (* uniqueness feature of new_prim *)
                  add constraint primary key (new_prim)
                                                  constraint idE1_#;
            end exec;
      for each foreign key in table E referencing table E1
          and representing relationship-type R
      do begin
            if u = 1 (* the old foreign key column R_old_prim was
                        mandatory *)
            then exec SQL
                        alter table E
                          drop constraint E_R_old_prim;
                  end exec;
            if y = 1 (* the old foreign key column R_old_prim was a unique
                        key *)
            then exec SQL
                        alter table E
                          drop constraint idE_#
                          add constraint unique (R_new_prim)
                                                  constraint idE_#;
                  end exec;
            exec SQL
                  (* we add the new foreign key constraint and remove the
                     old foreign key column *)
                  alter table E
                    add constraint foreign key (R_new_prim) references E1
                                                  constraint E1_#,
                      drop R_old_prim;
            end exec;
         end;
end; (* end of procedure *)
```

```
(* the program allows us to call the procedure 'Switch' with the correct
   arguments *)
if no unique key is specified
then begin
        exec SQL
          (* we create a technical identifier *)
          alter table E1
             add ID_E1 smallint  default 0 not null  constraint E1_ID_E1;
          (* we assign identifying values to that column *)
          declare c cursor for
            select ID_E1
              from E1
            for update of ID_E1 in E1;
          open c;
          fetch c;
        end exec;
        i:= 1;
        while SQLCODE = 0      (* the last item has not yet been treated *)
        do begin
             exec SQL
                 update E1
                   set ID_E1 = :i
                   where current of c;
                 fetch c;
             end exec;
             i:= i+1;
           end;
        exec SQL
           close c;
           (* we add the unique key feature to ID_E1 *)
           alter table E1
             add constraint unique (ID_E1) constraint idE1_#,
        end exec;
(* we operate the real switch *)
        call Switch(E1, a11, ID_E1);
      end
else (* a unique key is specified *)
      if the primary key of E1 is not a technical one
      then call Switch(E1, a11, a14)
      else call Switch(E1, ID_E1, a14);
```

No data is lost as we do not consider the information included in the technical identifier column ID_E1 as semantical data.

2.5.2.3. Program Extracts

Let us suppose we have switched primary key a11 with unique key a14.

Every select query which uses a foreign key referencing table E1 must be modified: we have to replace the foreign key.

```
- select ...
    from E
    where R_a11 = c
```

```
  select ...
    from E
    where R_a14 = d
```

```
- select ...
    from E1
    where a11 in( select R_a11
                  from E
                  where ... )
```

```
  select ...
    from E1
    where a14 in( select R_a14
                  from E
                  where ... )
```

A concrete example can be found in the modification switch_PK_unique in chapter 4 (see page 4-37). As we already said, it is not sufficient to change only the select queries. We must also review the application programs (for an example see page 4-41).

Chapter 6:

Introduction to the Modifications on the Basic ER Model

1. INTRODUCTION

In the previous chapters, we have seen the modifications of the Kernel. However, as the Kernel only allows restricted concepts, we have to enlarge it to what we call the Basic ER Model. This enlargement is illustrated in Figure 6-1.

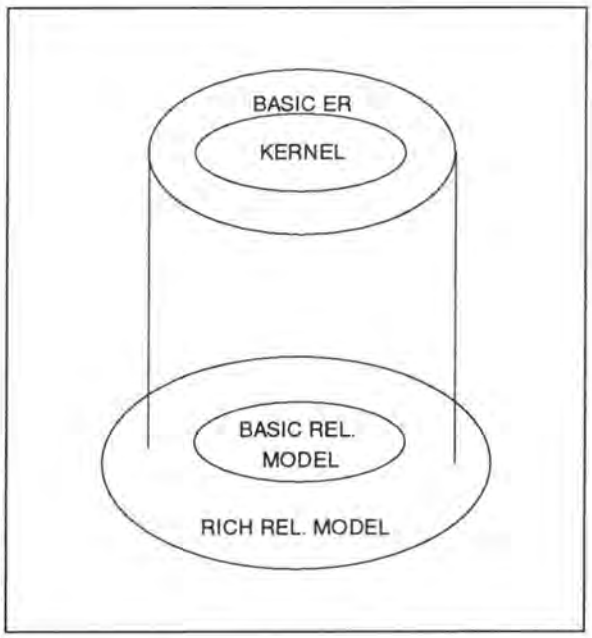


Figure 6 - 1 : Relation between the Basic ER model and the Rich Relational Model

Studying in detail the modifications on this model would be beyond the scope of our thesis and we will therefore only give some reflexions about them. In this chapter, we thus begin with a description of the Basic ER Model and we then briefly study the modifications on it. In this study, we consider the extension of existing objects, on the one hand, and the introduction of new objects, on the other hand. For both categories, we will consider new modifications and impacts on the existing ones.

2. DESCRIPTION OF THE BASIC ER MODEL

The Basic ER Model must be conceived in such a way that we are able to translate all its concepts into the Rich Relational Model. The latter groups all possible relational objects (tables, columns, foreign keys, primary keys, unique keys, check constraints, ...). Note that it should always be possible to translate any schema expressed in the Basic ER Model into the Rich Relational Model, but that the inverse translation may not always be possible.

We thus allow the following concepts for the Basic ER Model:

- *entity-types having at least one attribute and a primary key*
  If we would allow entity-types without a primary key, certain relationship-types connected to them could not be directly expressed in the relational model. We would have to add a technical identifier to the entity-types before translating these relationship-types into the relational model.

- *atomic and single-valued attributes, which can be optional or mandatory*
  This restriction is necessary as compound and/or multi-valued attributes must be decomposed and/or extracted before being translated into the relational model.

- *all the roles*
  Normally we would have to accept not only the cardinalities 0, 1, N, but all possible cardinalities. We will however not speak about them as they do not bring any new ideas and involve moreover unnecessary complications.

- *all functional*[1] *relationship-types*

  In addition to the relationship-types allowed in the Kernel (see page 3-2), we here also accept:

  - 1-1/1-N relationship-types (1-N/1-1 is symmetrical)
  - 0-1/1-N relationship-types (1-N/0-1 is symmetrical)
  - 1-1/1-1 relationship-types
  - recursive relationship-types

  We do not consider non functional relationship-types as they must be transformed into entity-types before being translated into the relational model.

- *identifiers of entity-types (the primary keys must be composed by mandatory attributes and/ or roles with minimum cardinality 1)*

  We only consider identifiers of entity-types as we do not need explicit identifiers for functional relationship-types. Indeed for these relationship-types the identifer(s) is (are) derivable from their maximum cardinalities.

- *constraints on roles: exclusion, inclusion, equality*
  on relationship-types: exclusion, inclusion, equality, functional dependency (FD)
  on attributes: coexistence, functional dependency (FD)

[1] A relationship-type is functional if it is binary, if it has no attributes and if it is not N-N.

3. STUDY OF THE MODIFICATIONS ON THE BASIC ER MODEL

3.1. INTRODUCTION

When comparing the description of the Basic ER Model (see page 6-2) to that one of the Kernel (see page 3-1), we can observe either an extension of the concepts of the Kernel or a creation of new objects. In both cases, new modifications and impacts on existing ones must be analysed. As we have already said, a detailed study of the modifications of the Basic ER model would be beyond the scope of our thesis. What is more, we will not try to aggregate the impacts of the different extensions of the existing objects and those of the new objects. Indeed, such an integration would be very difficult as one extension of an object will involve effects on the other extensions or on the new objects. We are here not able to study these mutual effects as we will not have made a detailed study.

To summarize, we will start with the extension of existing objects, then go over to the new objects.

3.2. EXTENSION OF EXISTING OBJECTS

Three major extensions can be observed in the Basic ER Model:

- allowing the minimum cardinality 1 everywhere and thus allowing
  - 1-1/1-N relationship-types (1-N/1-1 is symmetrical)
  - 0-1/1-N relationship-types (1-N/0-1 is symmetrical)
  - 1-1/1-1 relationship-types
- recursive relationship-types
- non mono-attribute identifiers

For each of these extensions, we will study the impacts on the existing modifications and analyse the new modifications.

3.2.1. Allowing the Minimum Cardinality 1 Everywhere

Before studying the impacts on the existing modifications, we will discuss how the minimum cardinality 1 is represented in SQL.

3.2.1.1. Representation in SQL

The minimum cardinality 1 of a 1-N role is represented in SQL by a check constraint. For example, let us consider the following relationship-type:

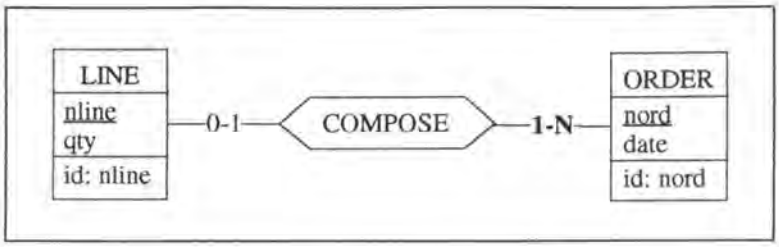


Figure 6 - 2 : A 0-1/1-N relationship-type on the conceptual level

In the relational model it is represented by:

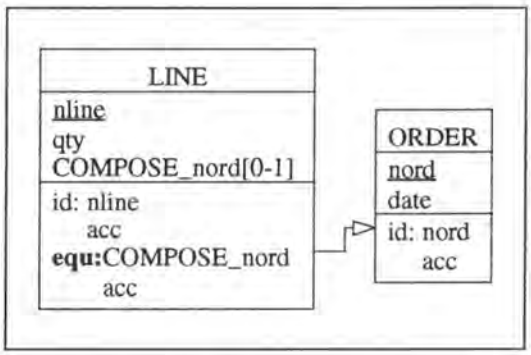


Figure 6 - 3 : An 0-1/1-N relationship-type on the logical level

In SQL-RDB the definition of the two tables would be:

```
create table ORDER
      ( nord            char(4)     not null    constraint O_nord,
        date            date        not null    constraint O_date,
        primary key (nord) constraint idORD1);


create table LINE
      ( nline             char(6)   not null    constraint L_nline,
        COMPOSE_nord      char(4),
        qty               integer   not null    constraint L_qty,
        primary key (nline) constraint idLIN1,
        foreign key (COMPOSE_nord) references ORDER constraint ORD1);


alter table ORDER
      add constraint check (nord in (select COMPOSE_nord from LINE))
                                                    constraint ch_nord1;
```

Note that we also need such a check constraint for the 1-1/1-1 relationship-types.

3.2.1.2. Impacts on the Existing Modifications

Allowing those minimum cardinalities has an impact on all[2] the modifications which handle either the foreign key representing the relationship-type, the minimum cardinality belongs to, or the primary key referenced by that foreign key. That is the case for the following modifications:

- *modifications of the entity-types:*	- rename_entity-type
- *modifications of the roles:*	- augment_max_card - decrease_min_card - decrease_max_card - augment_min_card
- *modifications of the attributes:*	- make_attr_mandatory
- *modifications of the identifiers:*	- switch_PK_unique

As we have already said, analysing in detail those impacts would be beyond the scope of our thesis. We therefore only study one modification in detail (augment_max_card) and give some indications for the others.

- In the modification *rename_entity-type* (see page 4- 10), we have now also to replace the check constraints referencing the entity-type that has to be renamed.

- Let us consider in detail *augment_max_card* (page 4-18):
  The precondition can be relaxed as we can now also accept the 1-1/1-1 → 1-1/1-N and the 0-1/1-1 → 0-1/1-N transformations.

  We will only examine the case 0-1/1-1 → 0-1/1-N. The other transformation is pretty similar.

  On the conceptual level the transformation is:

[2] We will not only consider the modifications described in Chapter 4, but also those included in Appendix 1.

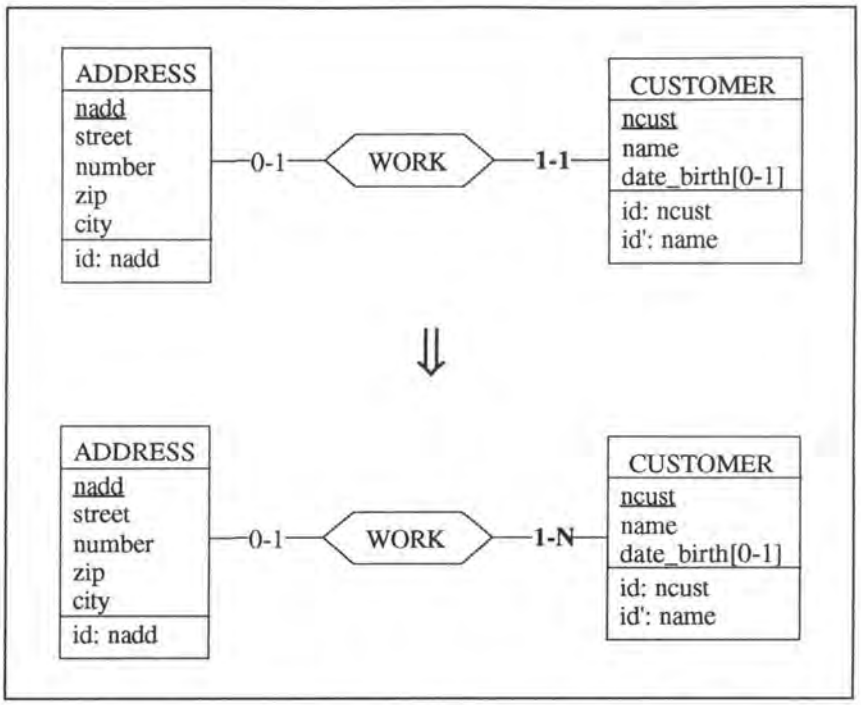


Figure 6 - 4 : Augmenting the maximum cardinality of a role to N on the conceptual level

On the logical level the transformation is:

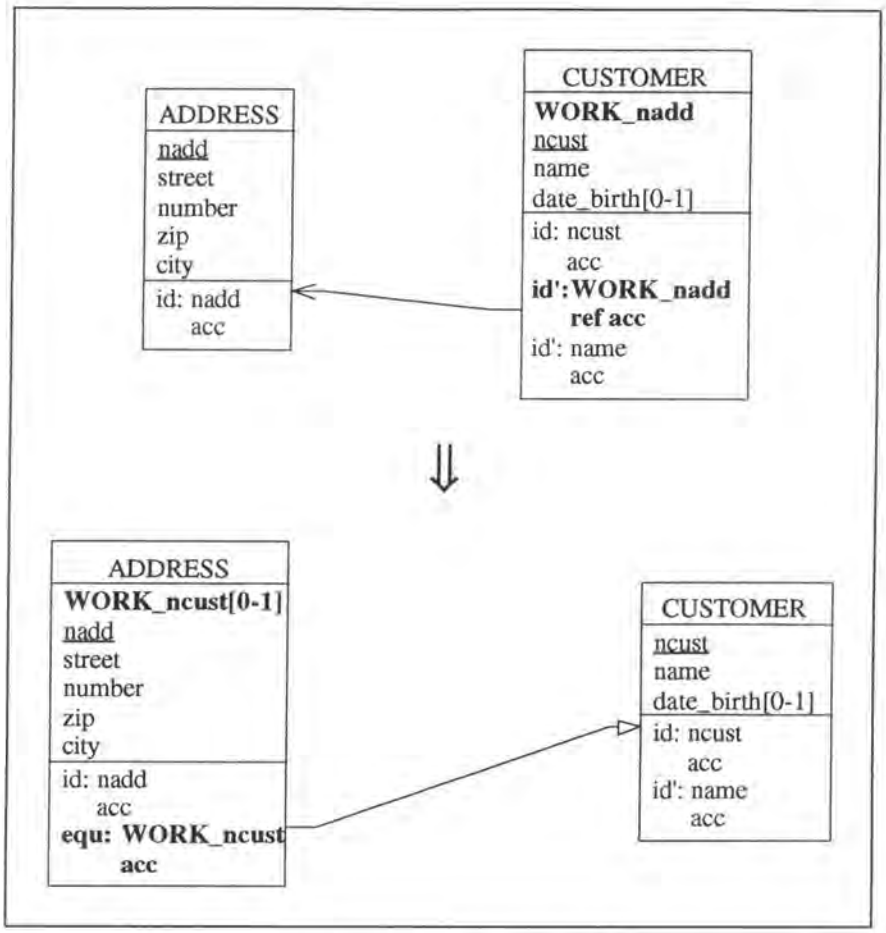


Figure 6 - 5 : Augmenting the maximum cardinality of a role to N on the logical level

The SQL description of the modification would become:

```
(* we will put in bold the additional operation *)
var cust: STRING[4];
    add: INTEGER;

exec SQL
      (* we create the new foreign key column *)
      alter table ADDRESS
        add WORK_ncust char(4);
      (* we copy the data representing relationship-type WORK from
         table CUSTOMER into table ADDRESS *)
      declare c cursor for
        select ncust, WORK_nadd
          from CUSTOMER;
      open c;
      fetch c into :cust, :add;
end exec;
while SQLCODE = 0       (* the last item has not yet been treated *)
do begin
      exec SQL
            update ADDRESS
              set WORK_ncust = :cust
              where nadd = :add;
            fetch c into :cust, :add;
      end exec;
     end;
```

```
exec SQL
      close c;
      (* we add and remove the necessary constraints *)
      alter table ADDRESS
        add constraint foreign key (WORK_ncust) references CUSTOMER
                                                constraint CUS1;
      alter table CUSTOMER
        drop constraint idCUS2,      (* we remove the old unique key
                                        feature *)
        drop constraint ADD1,        (* we remove the old foreign key
                                        feature *)
        drop constraint C_WORK_nadd,    (* we remove the mandatory
                                           feature     from     column
                                           WORK_nadd *)
        drop WORK_nadd,
        add constraint check (ncust in (select WORK_ncust
                                          from ADDRESS))
                          constraint ch_ncust1;
end exec;
```

Note that no data is lost as the data representing relationship-type WORK is only copied from table CUSTOMER into table ADDRESS.

Concerning the program extracts, the same remarks can be formulated as for the case 0-1/0-1 → 0-1/0-N where WORK is implemented by a foreign key in relation CUSTOMER (see page 4- 24).

- For the modification *decrease_min_card* (see page A1- 45), we have also to distinguish two new cases where we have to remove the check constraint representing the newly introduced minimum cardinality.

- *Decreasing the maximum cardinality* of a role to 1 (see page A1- 48) means adding the candidate key feature to the foreign key on the logical level. We have thus to remove rows with duplicate values for the foreign key which in its turn induces losses of values for a second foreign key of the entity-type the first foreign key belongs to. If there is a check constraint on the primary key referenced by the second foreign key then problems may occur! For example, let us suppose that we want to decrease the maximum cardinality of the 0-N role of relationship-type SPECIFY as indicated in Figure 6-6.

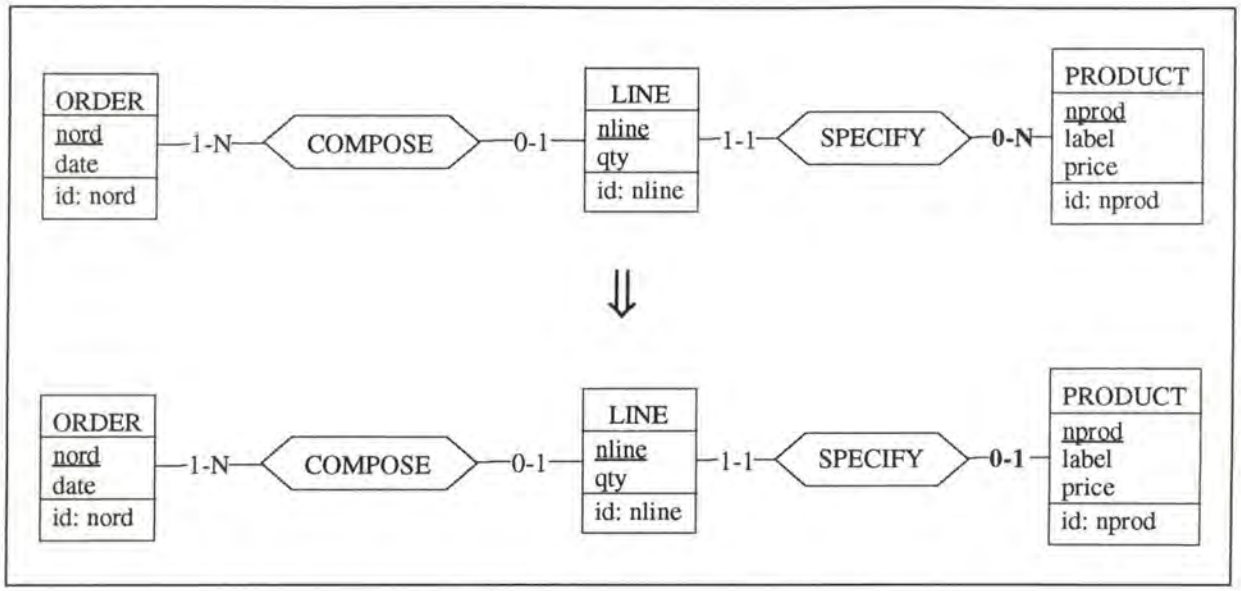


Figure 6 - 6 : Decreasing the maximum cardinality of a role to 1 on the conceptual level

On the logical level, this modification means adding the uniqueness feature to the foreign key SPECIFY_nprod, as depicted in Figure 6-7.

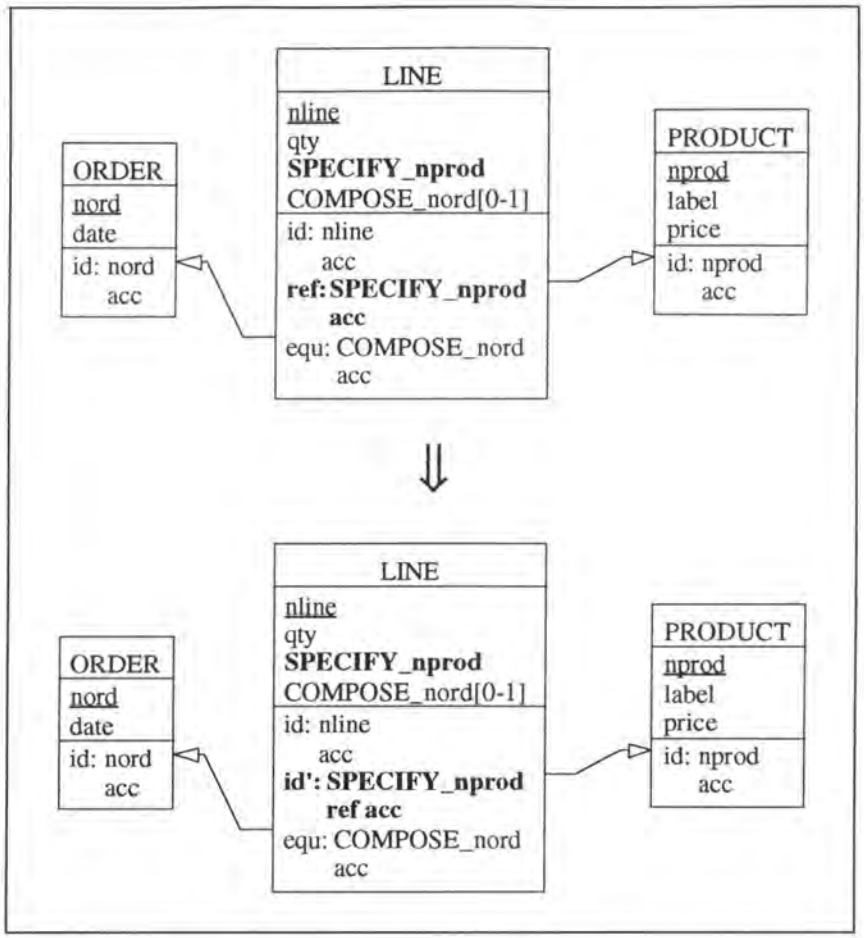


Figure 6 - 7 : Decreasing the maximum cardinality of a role to 1 on the logical level

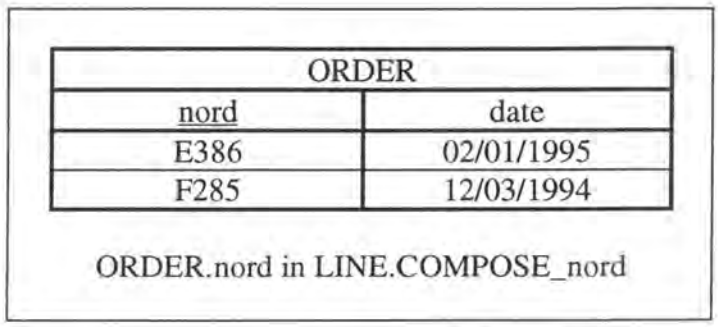
Let us suppose that we consider only the population of table LINE shown in Figure 6-8. As SPECIFY_nprod is now a unique key, we have to remove one of its two rows.

LINE			
nline	(COMPOSE_nord)	qty	SPECIFY_nprod
AB1234	E386	1000	AA110
GH2345	F285	1518	AA110

LINE.COMPOSE_nord in ORDER.nord
LINE.SPECIFY_nprod in PRODUCT.nprod

Figure 6 - 8 : A simplified table LINE

Let us suppose we remove the second row of table LINE. As we have a check constraint defined in table ORDER (see Figure 6-9), we have here also to drop the second row!



ORDER	
nord	date
E386	02/01/1995
F285	12/03/1994

ORDER.nord in LINE.COMPOSE_nord

Figure 6 - 9 : A simplified table ORDER

Note that this problem could occur again, if there were another table on which a check constraint applies on table ORDER.

- In the modification *augment_min_card* (see page A1-59), we have to distinguish three supplementary cases (1-1/0-1 → 1-1/1-1; 1-1/0-N → 1-1/1-N; 0-1/0-N → 0-1/1-N) where we have to express check constraints. In the existing cases, we are confronted to a similar problem as for the modification decrease_max_card. Here however, the problem occurs as we make a foreign key mandatory and thus remove the rows with a null value for that foreign key.

- In the modification *make_attr_mandatory* (see page 4-25), we are confronted to a similar problem as for the modification decrease_max_card, in case we remove the rows with a null value for the column which should be made mandatory. Here however, the problem occurs as we may loose values for foreign keys, if any, of the table the column belongs to.

- In the modification *switch_PK_unique* (see page 4-31), we have to adapt the check constraint to the new primary key.

3.2.1.3. New Modifications

Allowing minimum cardinality 1 everywhere induces new modifications:

those augmenting the semantics:	- add_1-1/1-N_rel-type - add_0-1/1-N_rel-type - add_1-1/1-1_rel-type
those decreasing the semantics:	- remove_1-1/1-N_rel-type - remove_0-1/1-N_rel-type - remove_1-1/1-1_rel-type
those preserving the semantics:	- rename_1-1/1-N_rel-type - rename_0-1/1-N_rel-type - rename_1-1/1-1_rel-type

These modifications are pretty similar to those allowed in the Kernel (see page A1-12), except that we also have to express check constraints.

3.2.2. Allowing Recursive Relationship-types

3.2.2.1. Impacts on the Existing Modifications

For the recursive relationship-types, essentially for those with cardinalities 0-1/0-1 or 1-1/1-1, at least one of their roles must have a name in order to be able to distinguish the roles from each other. This name[3] has to be used in the prefix of the foreign keys representing such recursive relationship-types and we have therefore to review all the modifications dealing with relationship-types.

3.2.2.2. New Modifications

In order to have a good administration of the recursive relationship-types, we would propose the following role modifications which preserve the semantics:

- add_role_name
- remove_role_name
- rename_role

- *Add_role_name*: This modification must add the name as a part of the prefix to the foreign key implementing the relationship-type, the role belongs to, and must rename the foreign key in all the constraints where it occurs.

- *Remove_role_name*: This modification must remove the name from the prefix of the foreign key implementing the relationship-type, the role belongs to, and must rename the foreign key in the constraints where it occurs. However, this modification should not be allowed for the recursive relationship-types where we cannot distinguish their roles by the cardinalities. It is the case of the 0-1/0-1 and the 1-1/1-1 recursive relationship-types.

- *Rename_role*: If the role is used in a prefix of a foreign key then it must be renamed in the foreign key and in the constraints where the foreign key occurs.

3.2.3. Allowing Non Mono-Attribute Identifiers

3.2.3.1. Impacts on the Existing Modifications

Allowing non mono-attribute identifiers has an impact on almost all modifications:

- *modifications of the entity-types:* - rename_entity-type

- *modifications of the relationship-types:* - add_x-1/0-v_rel-type
  - remove_x-1/0-v_rel-type
  - rename_x-1/0-v_rel-type

[3] If both roles have a name, only one of them is used in the prefix of the foreign key and we would thus loose the other one. In order to support a complete translation of such a recursive relationship-type into the Rich Relational Model, we would have to store the role name (which is not used in the prefix) in a semantic description provided by a CASE tool.

- *modifications of the roles:*	- augment_max_card - decrease_min_card - decrease_max_card - augment_min_card
- *modifications of the attributes:*	- make_attr_optional - remove_optional_attribute - remove_mandatory_attribute - make_attr_mandatory - all modifications changing the domain or the type of the attribute - rename_optional_attribute - rename_mandatory_attribute
- *modifications of the identifiers:*	- add_unique_feature - switch_PK_unique

Here too, we will only study one modification in detail (augment_max_card) and we will give some indications for the others.

- In the modification *rename_entity-type* (see page 4-10), we have to pay attention to the fact that the foreign key can be composed by several columns when adding the foreign keys to the new entity-type (in case we had no rename entity-type command).

- A similar remark can be formulated for all the *modifications on the relationship-types* (see page A1-12).

- Let us reconsider in detail *augment_max_card* (see page 4-18). We have to add a constraint to the precondition. We can only augment the maximum cardinality of a role (ro), if the other role of the relationship-type, ro belongs to, is not part of the primary key of the entity-type, connected to ro.

  We will only examine the case 0-1/0-1 → 0-1/0-N where WORK is implemented by a foreign key in relation CUSTOMER. (see case 3.4.2.2.2. page 4-23). On the conceptual level, the transformation is represented in Figure 6-10.

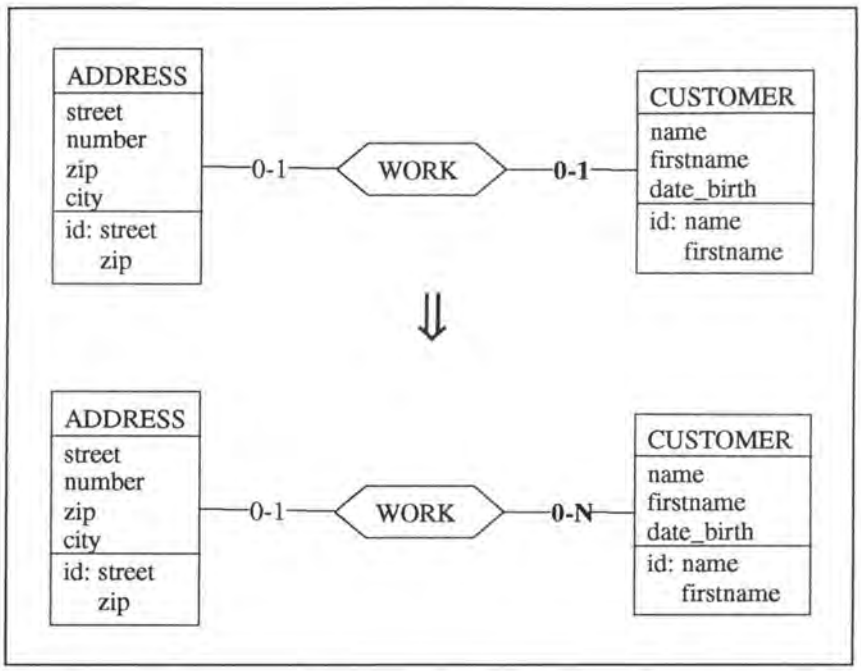


Figure 6 - 10 : Augmenting the maximum cardinality of a role to N on the conceptual level

On the logical level, the transformation is:

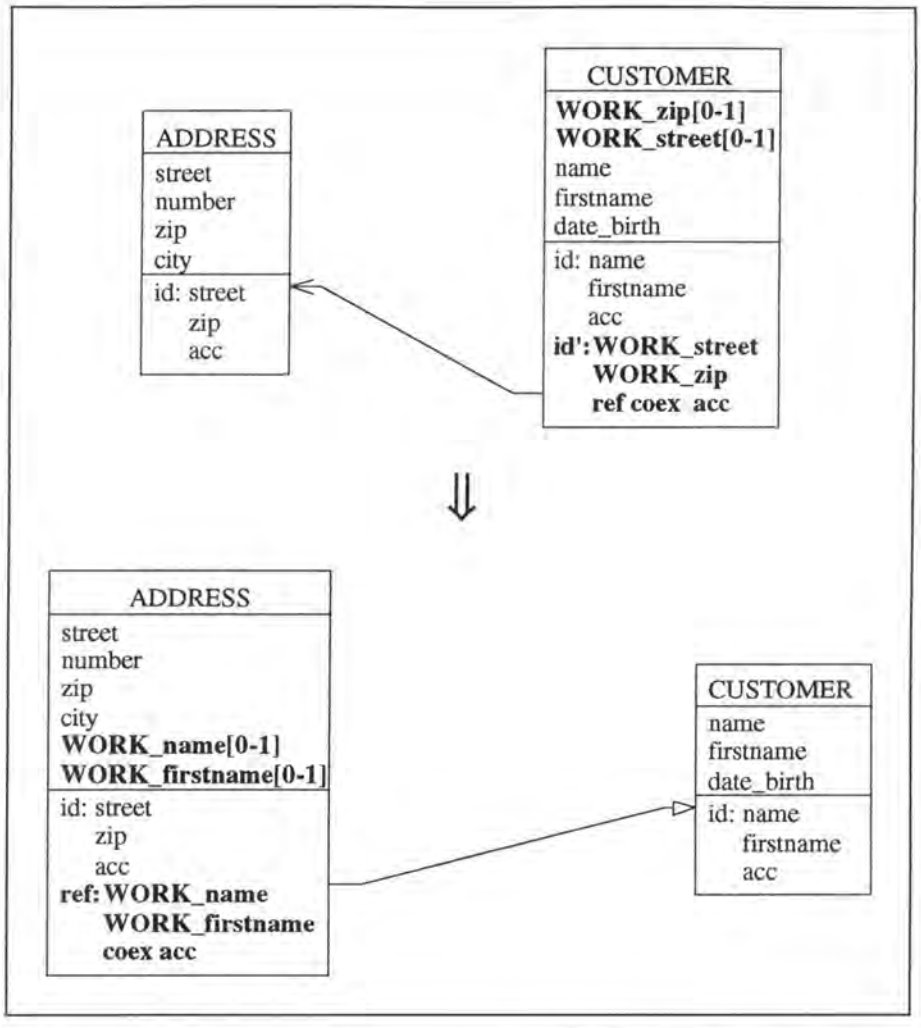


Figure 6 - 11 : Augmenting the maximum cardinality of a role to N on the logical level

Note:
As illustrated in Figure 6-11, there is a coexistence constraint between WORK_street and WORK_zip on the initial schema and between WORK_name and WORK_firstname on the final schema. However, as we have not yet treated the constraints (see New Objects page 6-19), we will not speak about these coexistence constraints until introducing the constraints.

The SQL description of the modification would become:

```
var name: STRING[12];
    firstname: STRING[20];
    street: STRING[20];
    zip: INTEGER;

exec SQL
      (* we create the new foreign key column *)
      alter table ADDRESS
        add WORK_name  char(12),
        add WORK_firstname  char(20);
      (* we copy the data representing the relationship-type WORK
         from table CUSTOMER into table ADDRESS *)
      declare c cursor for
```

```
        select name, firstname, WORK_street, WORK_zip
          from CUSTOMER
          where (WORK_street is not null) and (WORK_zip is not null);
      open c;
      fetch c into :name, :firstname, :street, :zip;
end exec;
while SQLCODE = 0 (* the last item has not yet been treated*)
do begin
     exec SQL
      update ADDRESS
        set WORK_name = :name,
            WORK_firstname = :firstname
        where street = :street and zip = :zip;
      fetch c into :name, :firstname, :street, :zip;
     end exec;
   end;
exec SQL
      (* we add and remove the necessary constraints *)
      alter table ADDRESS
        add constraint foreign key(WORK_name, WORK_firstname)
                              references CUSTOMER constraint CUS1;
      alter table CUSTOMER
        drop constraint idCUS2,      (* we remove the old unique key
                                        feature *)
        drop constraint ADD1,        (* we remove the old foreign key
                                        feature *)
        drop WORK_street,
        drop WORK_zip;
      close c;
end exec;
```

Note that no data is lost as the data representing relationship-type WORK is only copied from table CUSTOMER into table ADDRESS.

Concerning the program extracts, the same remarks can be formulated as for the case 0-1/0-1 → 0-1/0-N where WORK is implemented by a foreign key in relation CUSTOMER (see page 4-24) .

- *Decreasing the minimum cardinality* (see page A1-45) of a role means removing the mandatory constraint from the columns of the foreign key.

- In the modification *decrease_max_card* (see page A1-48), we have to pay attention to the fact that the primary key and thus the foreign keys can be composed by several columns.

- A similar remark can be formulated for the modification *augment_min_card* (see page A1-59).

- The precondition of *make_attr_optional* (see page A1-70) must be transformed into: " The attribute that should be made optional must not be **part of** a primary key. "

- As we do not want to manage here the fact to remove an element from an identifier and the loss of data it involves, we have to add to the precondition of the modification *remove_optional_attribute* (see page A1-75): "The attribute that should be removed, should not be part of a unique key."

- For similar reasons, the precondition of *remove_mandatory_attribute* (see page A1-76) must be reviewed as follows: "The attribute which should be removed, must not be **part of** an identifier and must not be the last attribute of the entity-type."

- In the modification *make_attr_mandatory* (see page 4-25), we have to pay attention to the fact that the primary key and thus the foreign keys can be composed by several columns.

- For all the *modifications changing the domain or the type of an attribute* (see page A1-71), the precondition must be reviewed as follows: "The attribute whose domain or type should be modified must not be **part of** an identifier."

- The operation *rename_optional_attribute* (see page A1-86) must distinguish two cases: the attribute is part of a unique key or not.

- The same remark as in the previous case can be formulated for the modification *rename_mandatory_attribute* (see page A1-88). As we do not want to rename the attribute in the foreign keys and in cascade (if the foreign key were part of the primary key of an entity-type), the precondition must be modified as follows: "The attribute which should be renamed must not be **part of** a primary key."

- In the modification *add_unique_feature* (see page A1-92), we have to pay attention to the fact that the primary key and thus the foreign keys can be composed by several columns. In addition, we have to check that the identifier that we want to add is not included in an already existing identifier. For example, the identifier 'name, firstname' should be refused if the identifier 'name, firstname, address' already exists.

- The precondition of *switch_PK_unique* (see page 4-31) must be modified as follows: "If a unique key is specified then **none of its components** must be optional." As for most of the other modifications, we must pay here also attention to the fact that the primary key and thus the foreign keys can be composed by several columns.

  Furthermore, allowing roles in the primary key involves the following problem:

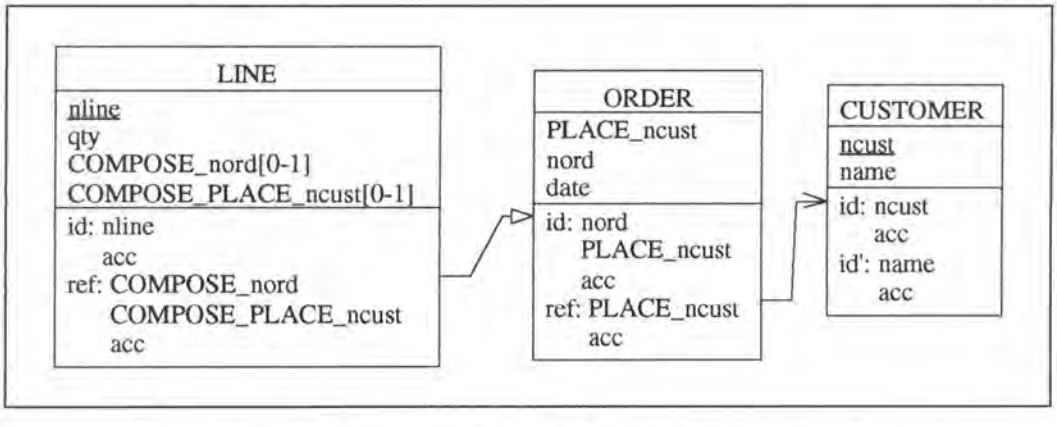


Figure 6 - 12 : Example where a foreign key is part of the primary key

If we want to switch the primary key ncust with the unique key name in CUSTOMER, we must not only replace the foreign key PLACE_ncust in ORDER by PLACE_name, but, as PLACE_ncust is part of the primary key of ORDER, we must also replace the foreign key column COMPOSE_PLACE_ncust in LINE by COMPOSE_PLACE_name. This problem

could occur recursively, for example, if COMPOSE_PLACE_ncust were part of the primary key of LINE.

3.2.3.2. New Modifications

As an identifier can now be composed by several attributes, adding and removing a component to/from an identifier are necessary operations. In order to avoid the above mentioned problem (see Figure 6-12), we would propose the operations only on unique keys:

modifications augmenting the semantics:	- add_attr_unique - add_role_unique
modifications decreasing the semantics:	- remove_attr_unique - remove_role_unique

- *Add_attr_unique*: This modification replaces the old unique key with a new one, the new key being the old one + the attribute that should be added.

- *Add_role_unique*: As a unique key of any relation E can only be composed by its own columns, we can add to the unique key only those roles, which are implemented by a foreign key in relation E.

- *Remove_attr_unique*: We have to check that the new identifier is not included in an already existing identifier. For example, if we have the identifiers 'name, firstname, address' and 'name, firstname, telephone' then it should be forbidden to remove either address or telephone of one of the two identifiers.

  Moreover, the fact to remove an attribute from an identifier involves not only loss of data in that relation, but can also involve loss of data in cascade. For an example of loss of data in cascade (though in a different context), the reader can refer to page A1-56.

- *Remove_role_unique*: The same remarks as in the modification remove_attr_unique can here be formulated.

3.3. NEW OBJECTS

The new objects are the constraints which can apply on three existing objects: the roles, the relationship-types and the attributes. We will briefly discuss their impacts on the existing modifications and speak about new modifications induced by the new objects.

Normally, modifications are only possible on objects on which no constraints apply. Thus, we have first to remove the constraints from the objects, then to execute the modifications on them and finally to restore the constraints on them. People who analyse in detail the impacts on modifications have the choice whether to require the non-existence of any constraints as a precondition or to take care of the constraints within the modifications.

The new objects do not only have an impact on existing modifications, but also induce new modifications. As it is not permitted to alter a constraint in SQL, we will only allow two basic operations for each constraint type: adding and removing a constraint. Adding a constraint decreases the semantics whereas removing a constraint augments it.

We will now briefly speak about the precondition of each of these operations. Note that the preconditions are written in such a way that there is no redundancy among the constraints.

- *Add_role_inclusion*: Both roles, the including and the included ones, must be connected to the same entity-type. If the including role has 0 as minimum cardinality then the included role must not have 1 as minimum cardinality. In addition, there must not be any exclusion constraint on these roles nor an inclusion constraint between their relationship-types.

- *Add_role_equality*: We will only note that an equality constraint is an inclusion constraint in both directions.

- *Add_role_exclusion*: The constraint must apply on two roles with minimum cardinalities 0 and these roles must be connected to the same entity-type. There must not be any inclusion constraint between these roles nor an inclusion or an exclusion between their relationship-types.

- *Add_rel_inclusion*: The constraint must apply on two relationship-types between the same entity-types and if any of the minimum cardinalities of the including relationship-type is 0 then the corresponding minimum cardinality of the included relationship-type must not be 1. In addition, there must not be any exclusion constraint on these relationship-types nor on their roles. Finally, there must not be any inclusion constraint among their roles.

- *Add_rel_equality*: We will only note that an equality constraint is an inclusion constraint in both directions.

- *Add_rel_exclusion*: The constraint must apply on two relationship-types between the same entity-types and there must not be any inclusion constraint between these relationship-types nor any exclusion on their roles.

- *Add_rel_FD*: The relationship-types on which the constraint applies must be connected to a same entity-type. There can only be one determined relationship-type and at least one of the determining relationship-types must have a role which has as minimum cardinality 1 and which is connected to the common entity-type. Finally, there must not be any exclusion constraint among the roles connected to the common entity-type.

- *Add_attr_coexist*: The attributes on which the constraint applies must be optional and must belong to the same entity-type.

- *Add_attr_FD*: The attributes on which the constraint applies must belong to the same entity-type. There can only be one determined attribute and at least one of the determining attributes must be mandatory. Note that adding a functional dependency (FD) having as determining attribute(s) an identifier would be redundant with the concept of identifier.

There are no preconditions on the remove_... operations.

In order to implement all these modifications, we have to add or remove check constraints in SQL.

Chapter 7:

Introduction to the Modifications on the Rich ER Model

1. INTRODUCTION

In the previous chapter, we have given a description of the Basic ER Model (see page 6-2) and some indications about the modifications on it. Generally, database schemas can however not be expressed in the Basic ER Model as it is too poor. We therefore have to introduce the Rich ER Model, which allows the most commonly used objects. In addition to the concepts already accepted in the Basic ER Model, we accept here:

- all entity-types
- compound attributes
- pure[1] multi-valued attributes
- non functional relationship-types:
  - n-ary relationship-types
  - N-N relationship-types
  - relationship-types with attributes
- identifiers of relationship-types
- functional dependencies on roles

Each schema expressed in this Rich ER Model must be translated into the Basic ER Model and each modification on a Rich ER schema (for a list of the modifications proposed on the Rich ER Model, see chapter 8 page 8-3) has an equivalent on the Basic ER schema. This equivalent can be composed by either one or more modifications of the Basic ER Model which is (are) then translated into the Rich Relational Model. The hierarchy of the different models is shown in Figure 7-1.

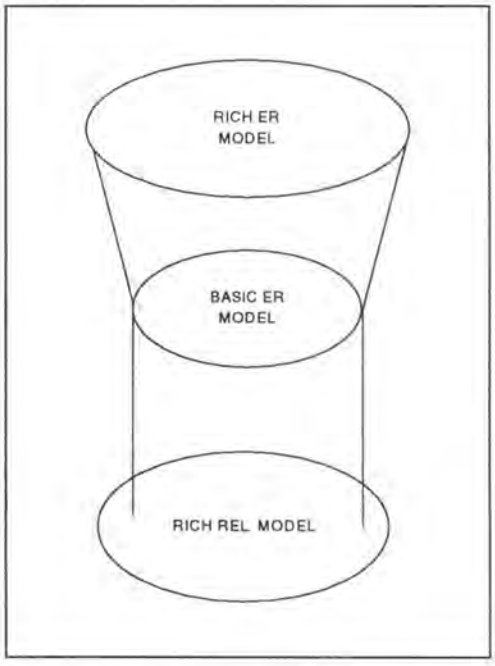


Figure 7 - 1 : The hierarchy of the different models

[1] A multi-valued attribute is said pure if its values for one instance of an entity-type are distinct.

2. MAPPING FROM THE RICH ER MODEL TO THE BASIC ER MODEL

In a first step, we will illustrate by examples how the new concepts used in schemas of the Rich E/R Model are mapped down to schemas in the Basic ER Model. In a further step, we will study on examples some modifications on the new concepts.

2.1. MAPPING OF THE NEW CONCEPTS

For each of the new objects, we will describe its mapping down to the Basic E/R Model.

2.1.1. Compound Attributes

There are three frequent techniques to represent compound attributes: decomposing them and extracting them either by instance or by value representation.

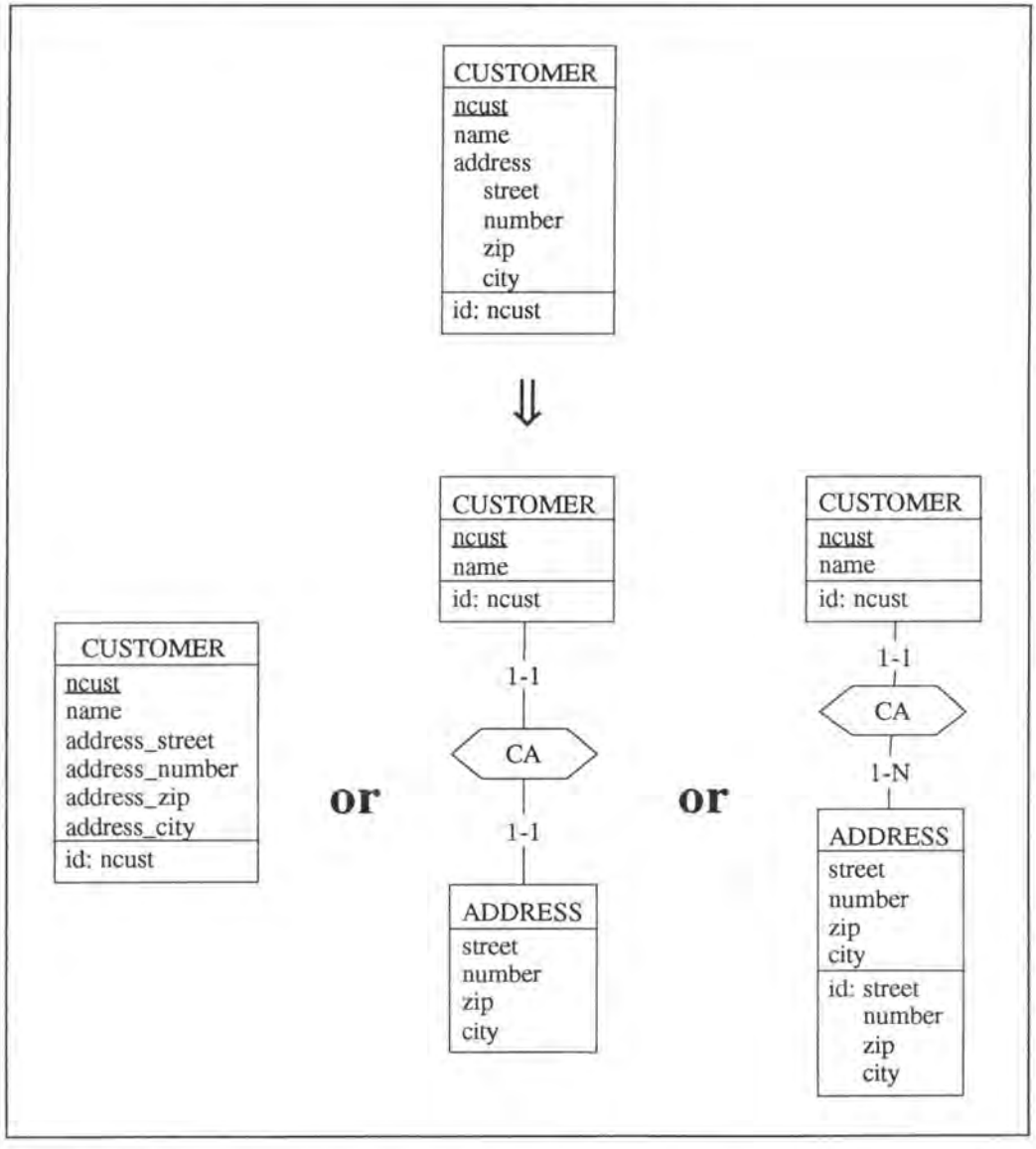


Figure 7 - 2 : Mapping a compound attribute of the Rich ER Model down to the Basic ER Model

2.1.2. Pure Multi-valued Attributes

The same three techniques can also be used to represent multi-valued attributes: decomposing the attributes and extracting them either by instance or by value representation.

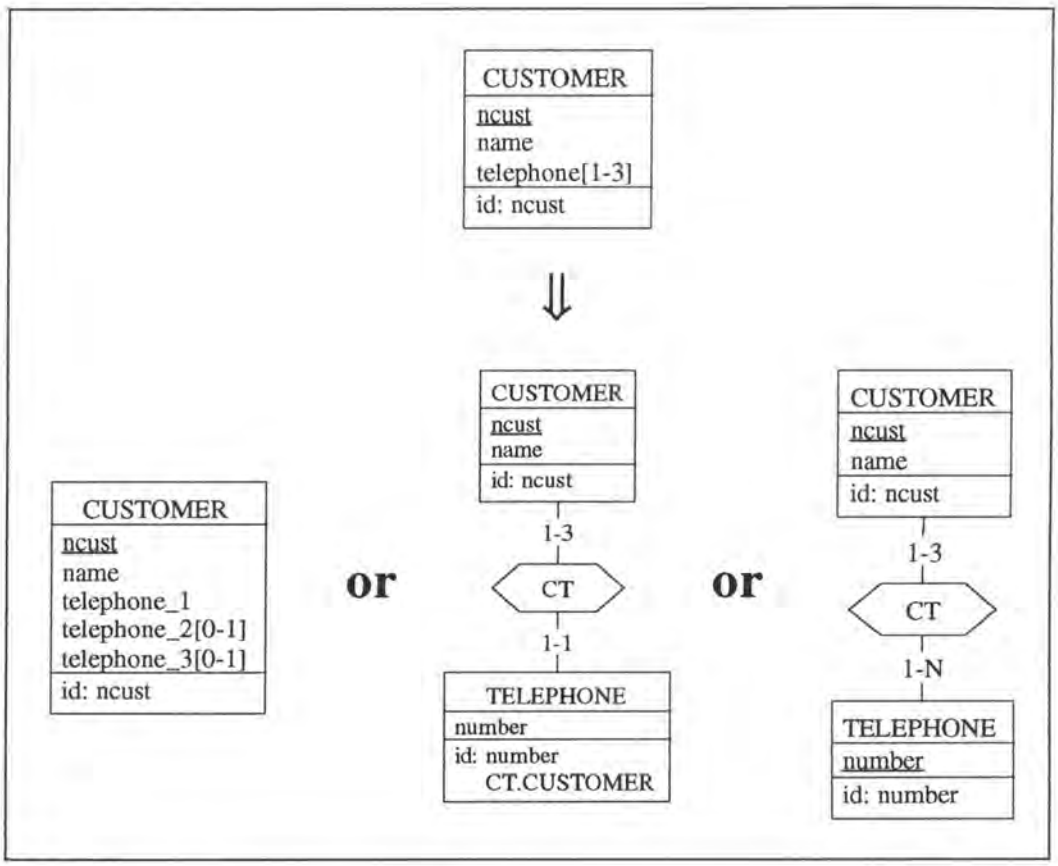


Figure 7 - 3 : Mapping a multi-valued attribute of the Rich ER Model down to the Basic ER Model

2.1.3. Non Functional Relationship-types

Generally, n-ary relationship-types must be transformed into entity-types. We will illustrate such a transformation by a ternary relationship-type with attributes.

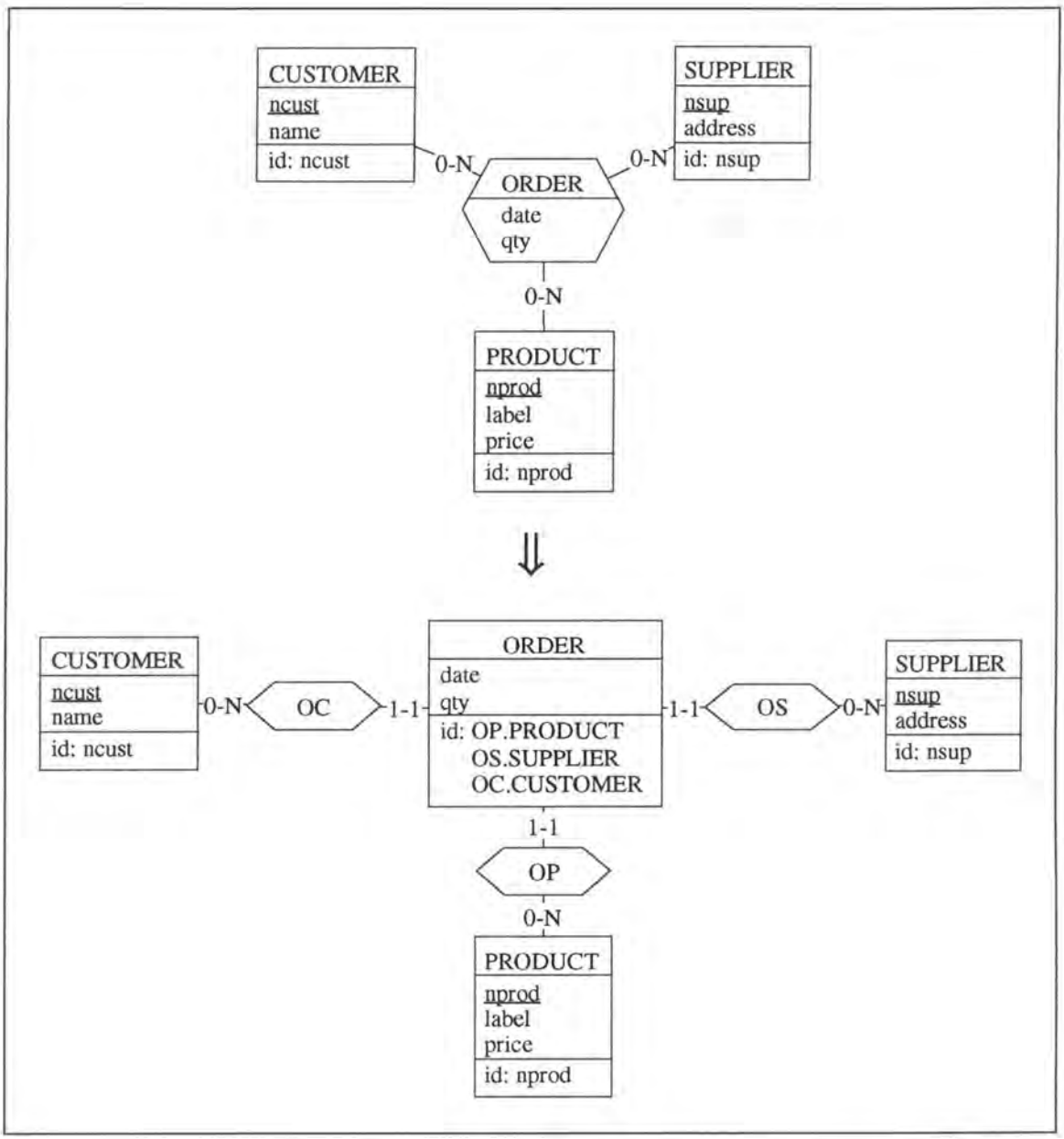


Figure 7 - 4 : Mapping a ternary relationship-type of the Rich ER Model into the Basic ER Model

The N-N relationship-types and the relationship-types with attributes are transformed in a similar way.

2.1.4. Identifiers of Relationship-types

As we now consider non functional relationship-types, we have also to consider identifiers of relationship-types. Let us reexamine the example of Figure 7-4, this time introducing however an explicit identifier

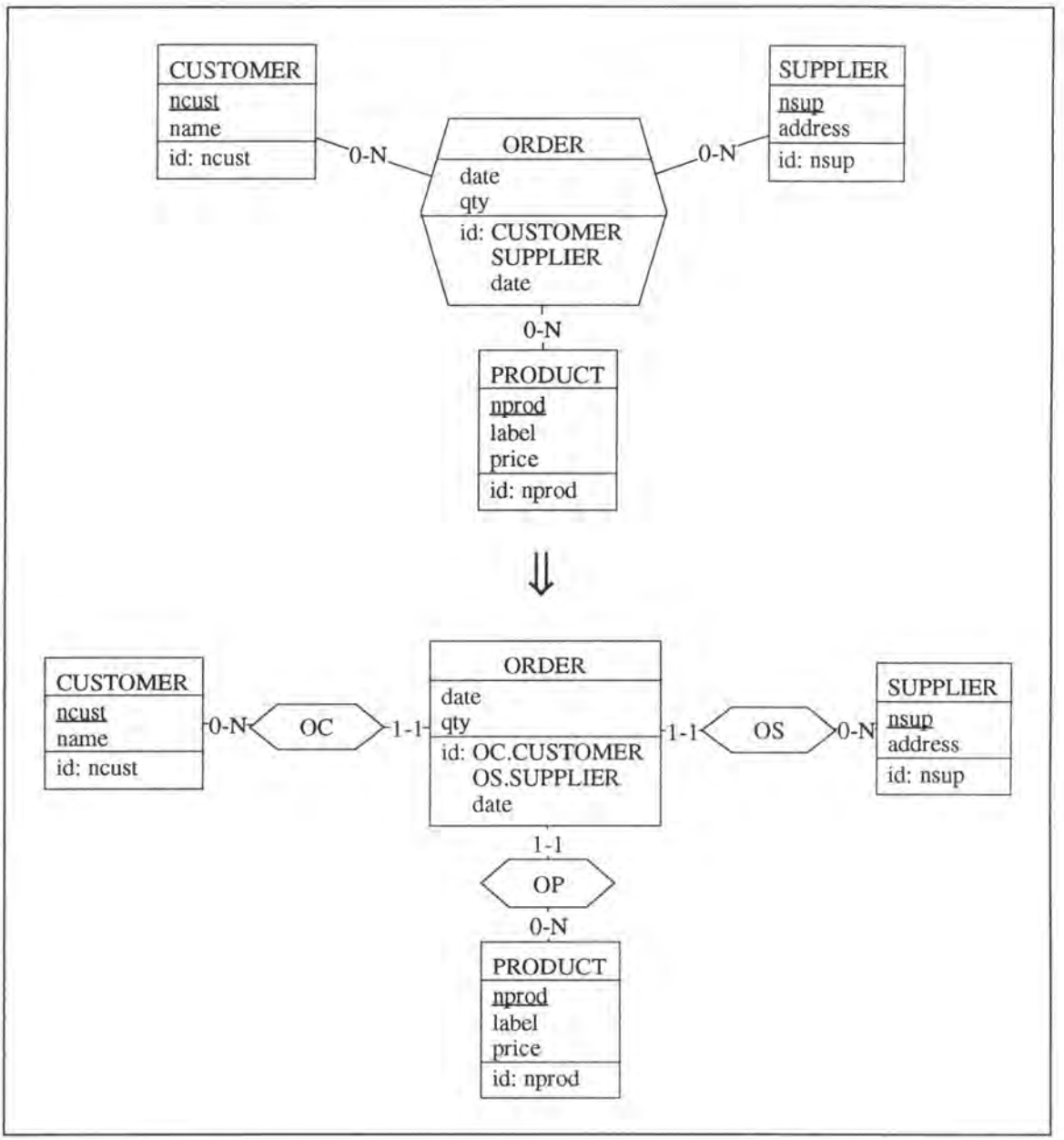


Figure 7 - 5 : Mapping an identifier of a ternary relationship-type of the Rich ER Model into the Basic ER Model

2.1.5. Functional Dependencies on Roles

A first idea to map into the Basic ER Model a ternary relationship-type on which a functional dependency applies would be to decompose it along the functional dependency.

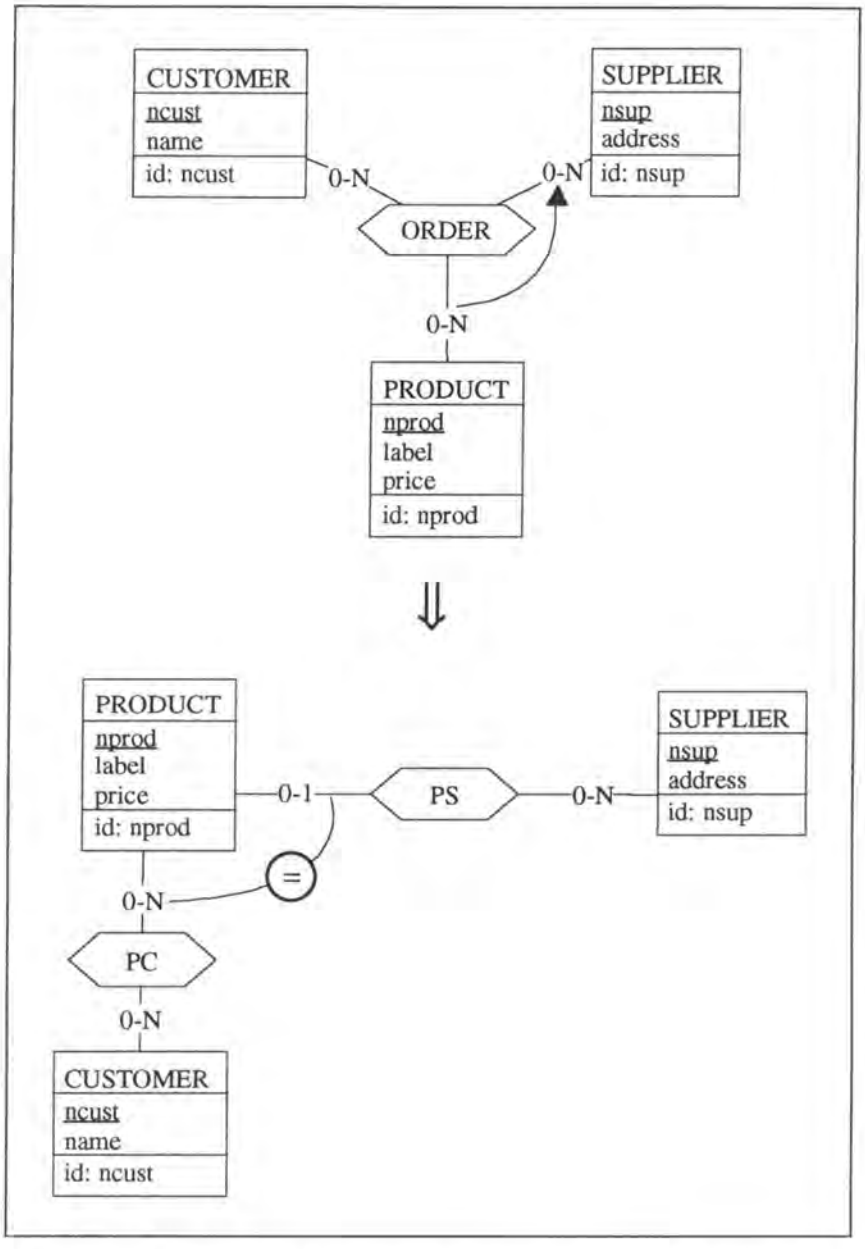


Figure 7 - 6 : Mapping a functional dependency of a ternary relationship-type of the Rich ER Model into the Basic ER Model, a first approach

This way of mapping does however not work very well in all the cases. Indeed, let us consider the non functional relationship-type ORDER which has been transformed as indicated in Figure 7-4 (see page 7-4). If we would add a functional dependency between the roles played by PRODUCT and SUPPLIER, then mapping the transformation, as indicated in Figure 7-6, would require a.o. the creation of a new table and the 'copy' of the data representing relationship-type ORDER into it. This copy operation is very time-consuming (especially if

table ORDER is large). We thus recommend, as illustrated in Figure 7-7, to keep the ternary relationship-type ORDER as an entity-type and to map the functional dependency on roles into a functional dependency on relationship-types. Note however that this alternative induces an unnormalised conceptual and thus relational schema.

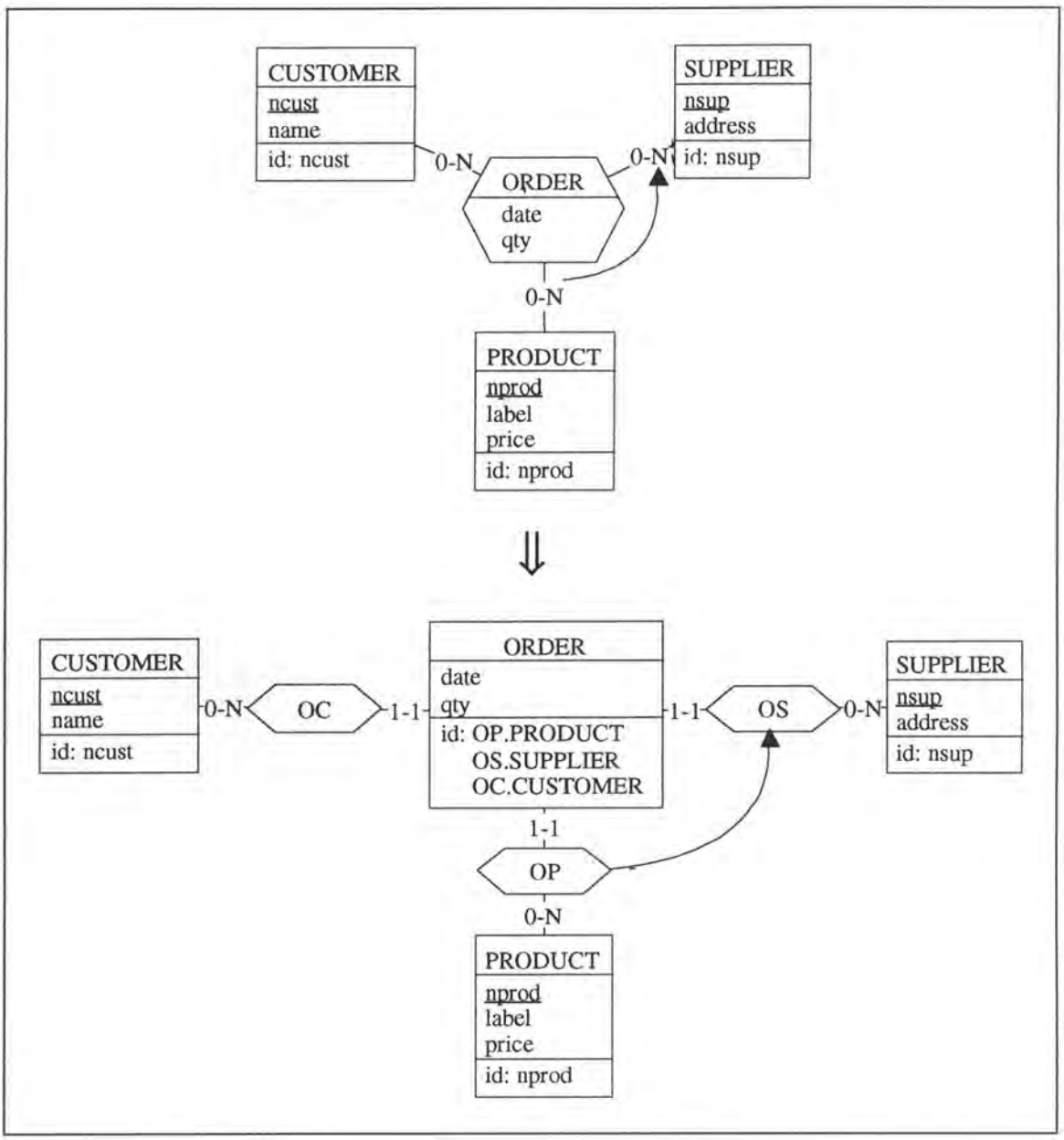


Figure 7 - 7 : Mapping a functional dependency of a ternary relationship-type of the Rich ER Model into the Basic ER Model

2.2. MAPPING OF THE MODIFICATIONS

We will here only study briefly two examples of how to map a modification on a schema of the Rich ER Model onto the corresponding schema of the Basic ER Model. We will first show how to make a compound attribute mandatory. In a further step, we will add a first attribute to a functional relationship-type.

2.2.1. Making a Compound Attribute Mandatory

Let us illustrate the modification by Figure 7-8.

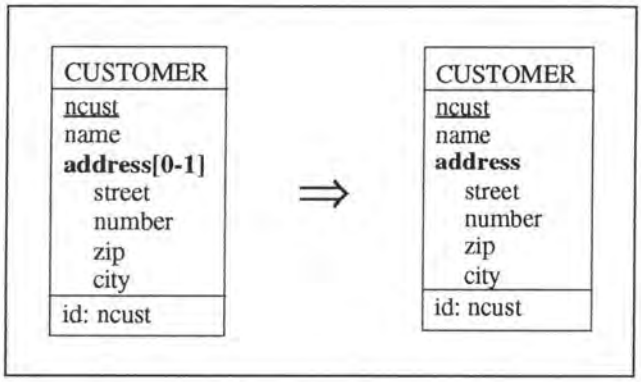


Figure 7 - 8 : Making a compound attribute mandatory in the Rich ER Model

In order to study the mapping of the modification, we have to know how the compound attribute has been represented in the Basic ER Model. As we have seen in Figure 7-2 (see page 7-2), we have to distinguish three cases:

- decomposition of the attribute
- extraction of the attribute by instance representation
- extraction of the attribute by value representation

2.2.1.1. Decomposition of the Attribute

As shown in Figure 7-9, we have first to remove the coexistence constraint from the decomposed attributes and then make each of them mandatory.

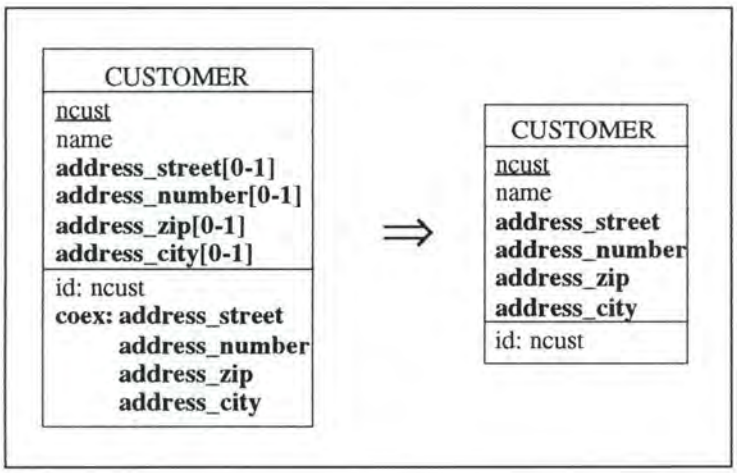


Figure 7 - 9 : Making a compound attribute mandatory in the Basic ER Model where it is represented by decomposition

2.2.1.2. Extraction of the Attribute by Instance Representation

As shown in Figure 7-10, we have in this case only to increase to 1 the minimum cardinality of the 0-1 role of relationship-type CA.

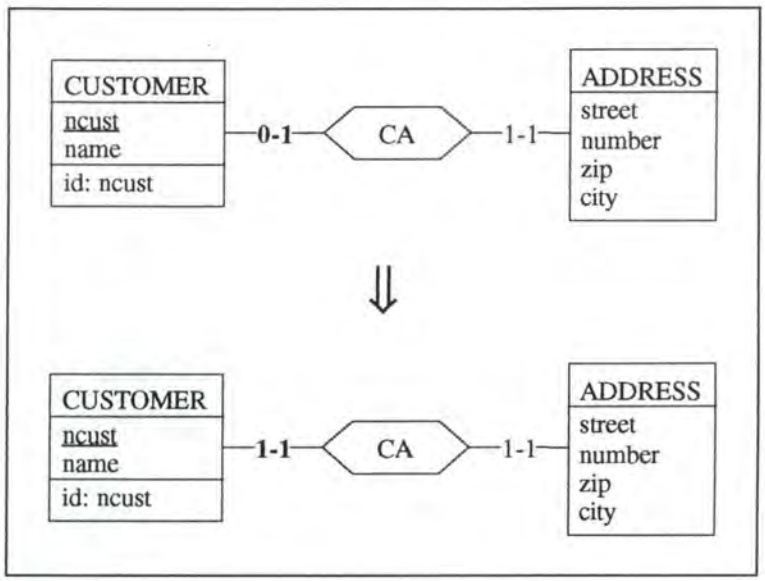


Figure 7 - 10 : Making a compound attribute mandatory in the Basic ER Model where it is extracted by instance representation

2.2.1.3. Extraction of the Attribute by Value Representation

The mapping is exactly the same as in the previous case. The modification in the Basic ER Model is represented in Figure 7-11.

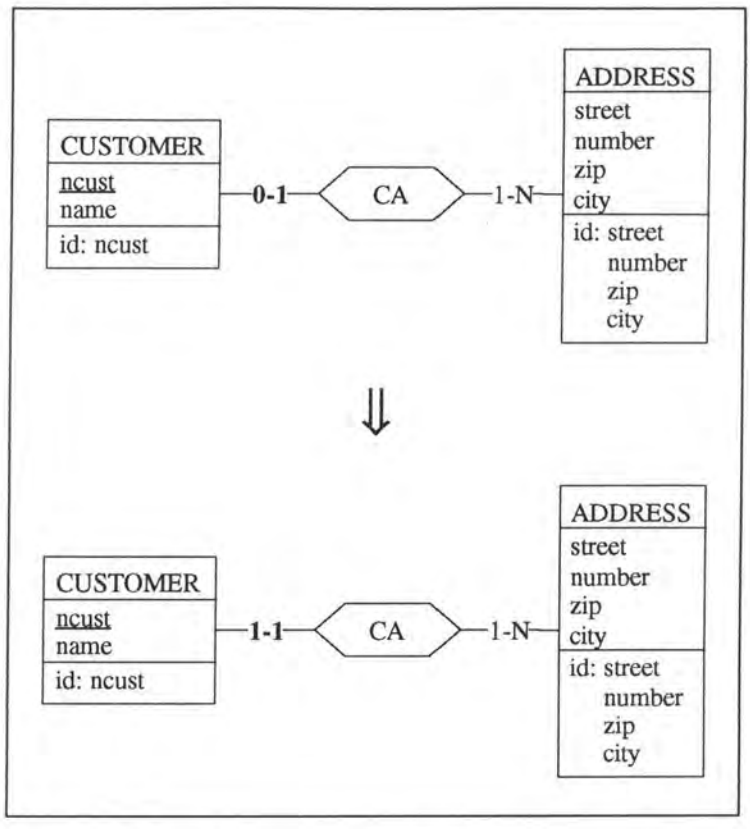


Figure 7 - 11 : Making a compound attribute mandatory in the Basic ER Model where it is extracted by value representation

2.2.2. Adding a First Attribute to a Functional Relationship-type

Let us consider the modification where we want to add a first attribute qty to the functional relationship-type MANUFACTURE. This situation is depicted in Figure 7-12.

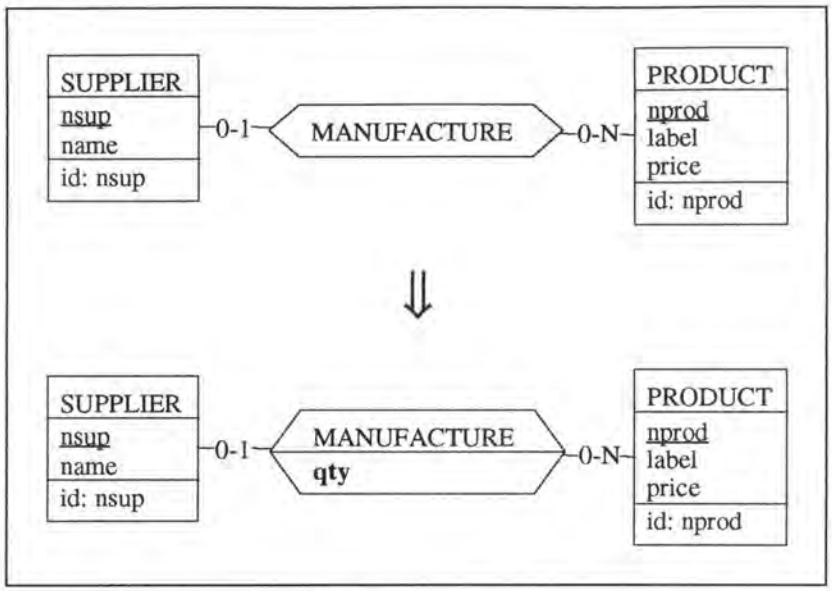


Figure 7 - 12 : Adding a first attribute to a functional relationship-type in the Rich ER Model

In order to map the modification down to the Basic ER Model (see Figure 7-13), a first idea would be to remove the relationship-type MANUFACTURE, then to add an entity-type MANUFACTURE with the attribute qty and finally to add two relationship-types MS and MP.

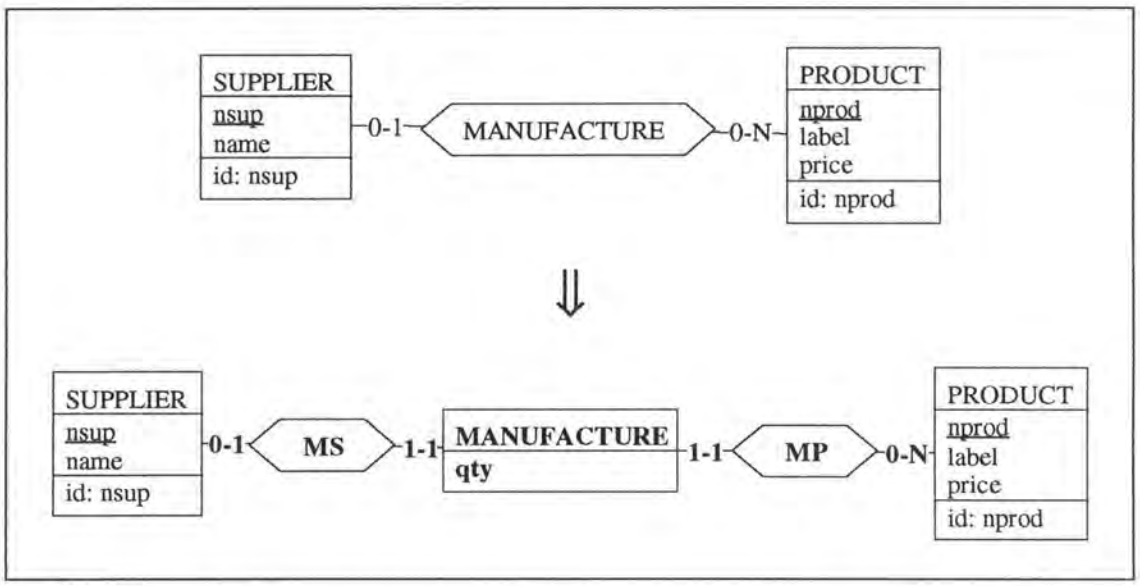


Figure 7 - 13 : Adding a first attribute to a functional relationship-type in the Basic ER Model

This way of proceeding does however not work if we consider populated databases. Indeed, none of the modifications in the Basic ER Model allows us to copy the data representing relationship-type MANUFACTURE into the entity-type MANUFACTURE and we would thus loose all this data. We are confronted here to the problem that mapping modifications of the Rich ER Model down to the Basic ER Model requires supplementary modifcations on the Basic ER Model. In our example, we would need the modification *'transform_rel-type→entity-type'*. The mapping of the modification 'add an attribute' (of the Rich ER Model) would now

consist in executing first the 'transform_rel-type→entity-type' and then the 'add_mandatory_attribute' operations.

Before concluding this chapter, we will study the modification 'transform_0-1/0-N_rel-type→ entity-type' that we have to add to the Basic ER Model.

On the conceptual level, the modification is illustrated in Figure 7-14[2].

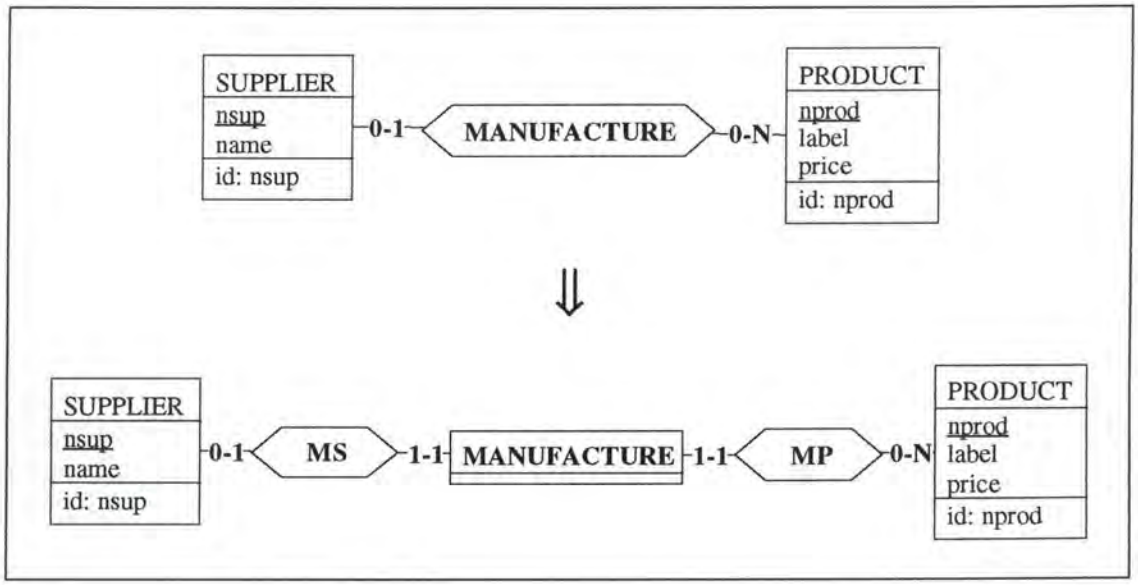


Figure 7 - 14 : Transforming a 0-1/0-N relationship-type into an entity-type on the conceptual level

On the logical level, we have to remove the foreign key representing relationship-type MANUFACTURE from relation SUPPLIER, to add relation MANUFACTURE, to add the foreign keys representing relationship-types MS and MP and to add the primary key feature to the foreign key column MS_nsup in relation MANUFACTURE.

[2] It can be observed that the entity-type MANUFACTURE has no attributes which is contrary to the description of the Basic ER Model. We will however tolerate this situation as the modification 'transform_0-1/0-N_rel-type→entity-type' is always followed by a modification which adds an attribute to the concerned entity-type.

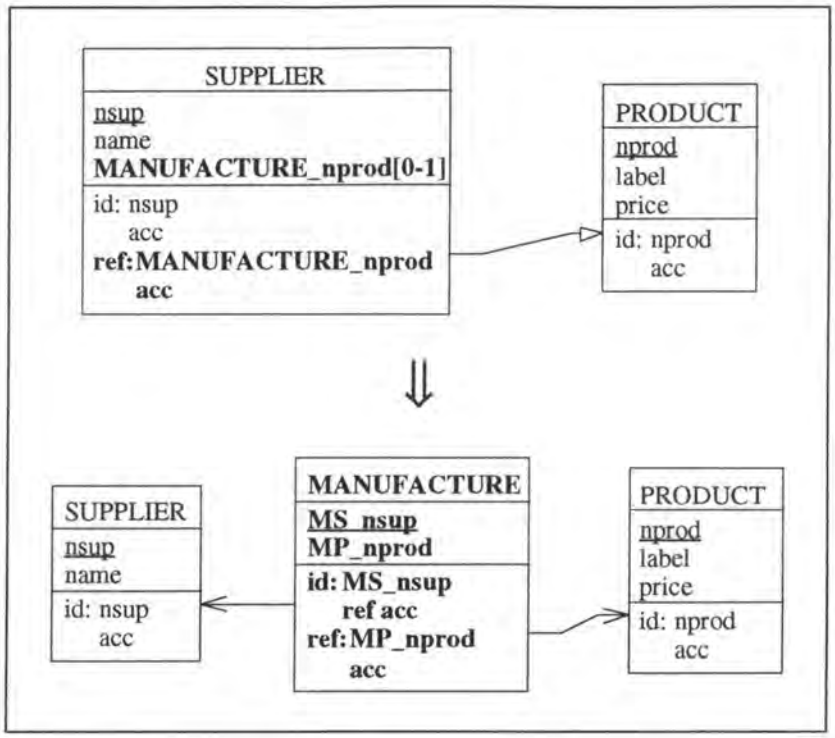


Figure 7 - 15 : Transforming a 0-1/0-N relationship-type into an entity-type on the logical level

The SQL description of the modification would be:

```
var sup, prod: char(5);

exec SQL
      (* we drop the foreign key constraint *)
      alter table SUPPLIER
        drop constraint PRO1;
      (* we create the new table with the necessary foreign keys *)
      create table MANUFACTURE
            ( MS_nsup   char(5)  not null constraint M_MS_nsup,
              MP_nprod  char(5)  not null constraint M_MP_nprod,
              primary key (MS_nsup) constraint idMAN1,
              foreign key (MS_nsup) references SUPPLIER constraint SUP1,
              foreign key (MP_nprod) references PRODUCT constraint PRO1);
      (* we copy the data representing relationship-type MANUFACTURE from
         table SUPPLIER into table MANUFACTURE *)
      declare c cursor for
        select nsup, MANUFACTURE_nprod
          from SUPPLIER
          where MANUFACTURE_nprod is not null;
      open c;
      fetch c into :sup, :prod;
end exec;
while SQLCODE = 0 (* the last item has not yet been treated *)
do begin
      exec SQL
            insert into MANUFACTURE
              values :sup, :prod;
            fetch c into :sup, :prod;
      end exec;
   end;
exec SQL
      close c;
      (* we remove the old foreign key column *)
```

```
      alter table SUPPLIER
        drop MANUFACTURE_nprod;
end exec;
```

Note that no data is lost as the data representing relationship-type MANUFACTURE is copied from table SUPPLIER into table MANUFACTURE.

The program extracts referencing the foreign key representing the relationship-type MANUFACTURE must be reviewed. A possible modification for the select queries would be:

```
select name
  from SUPPLIER
  where MANUFACTURE_nprod in ( select nprod
                               from PRODUCT
                               where label = 'christmas tree')
```

```
select name
  from SUPPLIER
  where nsup in ( select MS_sup
                  from MANUFACTURE
                  where MP_nprod in ( select nprod
                                      from PRODUCT
                                      where label = 'christmas tree')).
```

Chapter 8:

Integration into a CASE Tool

1. INTRODUCTION

As we have underlined in chapter 1, database evolution has become more and more important during the last few years. Economically seen, database evolution induces enormous costs and the demand for CASE tools offering evolution facilities increases steadily.

In order to integrate into the DB-MAIN[1] CASE tool the previously studied modifications, we will first describe the objective of the DB-MAIN tool and then indicate its main aspects and components supporting database evolution.

2. THE DB-MAIN TOOL

2.1. OBJECTIVE

DB-MAIN is a generic CASE tool dedicated to database application engineering and in particular to database design, reverse engineering and maintenance. The objective is to study in a uniform framework the problems which arise when the requirements of database applications evolve. The CASE tool is a preliminary version of the future Database Applications Maintenance Assistant which is the ultimate goal of the development activity [HAI94b page 2].

As we can see, the objective of our thesis matches well with the one of the DB-MAIN tool. We will now analyse how the different components of the DB-MAIN tool can be used for the integration of our modifications. We will therefore essentially refer to [HAI95a].

2.2. COMPONENTS OF THE DB-MAIN TOOL

As a large-scope CASE tool, DB-MAIN includes the usual functions needed in database analysis and design, e.g. entry, browsing, management and modifications of specifications, as well as code and report generation. However, the following sections will concentrate on the main aspects and components of the tool which are directly related to database evolution activities, namely:

- the data structure representation model, and the repository
- the modification toolboxes
- the user interface
- text analyser and name processor
- the specialized assistants
- functional extensibility
- methodological control and guidance

[1] Standing for DataBase MAINtenance.

2.2.1. The DB-MAIN Specification Model and Repository

The repository collects all the information related to a project and comprises thus three classes of information:

- a structured collection of specifications, i.e. schemas and texts
- the specification of the methodology followed to conduct the project
- the history (or trace) of the project

A schema is a description of the data structures to be processed, while a text is any textual material generated or analysed during the project (e.g. a program or an SQL script). A project usually comprises many schemas and the schemas of a project are linked through specific relationships: the hierarchical links on the one hand and the historical ones on the other hand. By hierarchical links we mean the kind of link that exists between a conceptual schema and the corresponding logical one, whereas the historical ones stand for the successive versions of the conceptual schemas. Note that we only regard historical relationships among conceptual schemas as, for the moment, we only consider modifications on the conceptual level.

In order to represent those concepts, the repository uses a unique model for representing the schemas of a database, model which is based upon the ER model. We have thus to distinguish three types of constructs:

- *conceptual constructs* such as entity-types, attributes and identifiers
- *logical constructs* such as referential constraints
- *physical constructs* such as access keys

2.2.2. The Modification Toolkit

DB-MAIN proposes a three-level modification toolset that can be used freely, according to the skill of the user and the complexity of the problem to be solved. We thus distinguish:

- *elementary modifications:*
  One modification is applied to one object; with these tools, the user keeps full control on the schema modification since similar problems can be solved by different modifications; e.g. an entity-type can be transformed in differents ways.

- *global modifications:*
  One modification is applied to all the relevant objects of a schema. This toolset is controlled by the Modification Assistant (see 2.2.4. page 8-6). Example: replace all multi-valued attributes by entity-types + many-to-one relationship-types.

- *model-driven modifications:*
  All the constructs of a schema that do not comply with a given model (e.g. a specific normalised model) are modified; these modifications require little control from the user. Note that the analyst can build its own model driven modifications through scripting facilities of the Modification Assistant.

We will now give the most important modifications for database evolution that we would propose to the DB-MAIN users. Remember that these modifications apply on the objects of the Rich ER Model only.

Modifications on:

- Entity-type
  - Add
  - Remove
  - Rename
  - Transform
    - → Relationship-type
    - → Attribute

- Relationship-type
  - Add
  - Remove
  - Rename
  - Transform
    - → Entity-type

- Role
  - Add
  - Remove
  - Rename
  - Change cardinality
    - Max card
    - Min card

- Attribute
  - Add
  - Remove
  - Rename
  - Change domain
    - Extend
    - Restrict
  - Change type
    - Character
    - Float
    - Integer
    - Date
  - Change property
    - Make optional
    - Make mandatory
    - Make single-valued
    - Make multi-valued
    - Compose
    - Decompose
  - Transform
    - → Entity-type

- Identifier
  - Add unique
  - Remove unique
  - Switch PK ↔ unique

Change unique
Add attribute
Remove attribute
Add role
Remove role

2.2.3. The User Interface

The user interaction operates through a fairly standard GUI. However, interacting with the specifications exhibits some original options which deserve being discussed.

Browsing through, processing and analysing large schemas and texts require an adequate presentation. It quickly appears that more than one way of viewing them is necessary. For instance, a graphical representation of a schema allows an easy detection of certain structural patterns. DB-MAIN offers six ways of presenting a schema:

Text-based views:
- *compact*: sorted list of the names of the entity-types and relationship-types
- *standard*: simple structured list of the entity-types (name + attributes + constraints) and relationship-types (name + roles + attributes + constraints)
- *extended*: cfr standard with more details (data types, short names and so on
- *sorted*: the name of all the objects are presented in a sorted list

Graphical views:
- *compact*: only entity-types, relationship-types and roles are shown
- *standard*: same contents as text-based standard view

Switching from one view to another one is immediate, and any object selected in a view is still current when another view is chosen. Any relevant modification can be applied on an object, whatever the view through which it is presented. In addition, the text-based views allows navigating from entity-types to relationship-types and conversely through hypertext links.

As the modifications have an impact on the logical level, on the SQL description , on the data and on the applications programs, the user must be able to view these different components. We thus propose the following perceptions:

Rich ER Schema
Logical Schema
SQL Description
Data
Application Programs

Note however that the modifications can only be applied on the Rich ER Model. Let us recall here that for each modification we treat the impact on the data individually. An optimisation would be to allow to handle the impact on the data for a whole group of modifications applying on the same object. For instance, if the user wants to remove two attributes from an entity-type, then we would access twice the table representing the concerned entity-type. It would however be better (in terms of time and ressources) to treat these modifications in a single process.

Let us consider the operations a user has to execute in order to add a 1-1/0-1 relationship-type between two existing entity-types:

- choose menu 'Modification' from the action bar
- choose 'Rel-type' from that pull-down menu
- choose 'Add' in that cascading menu

The following dialog box appears then:

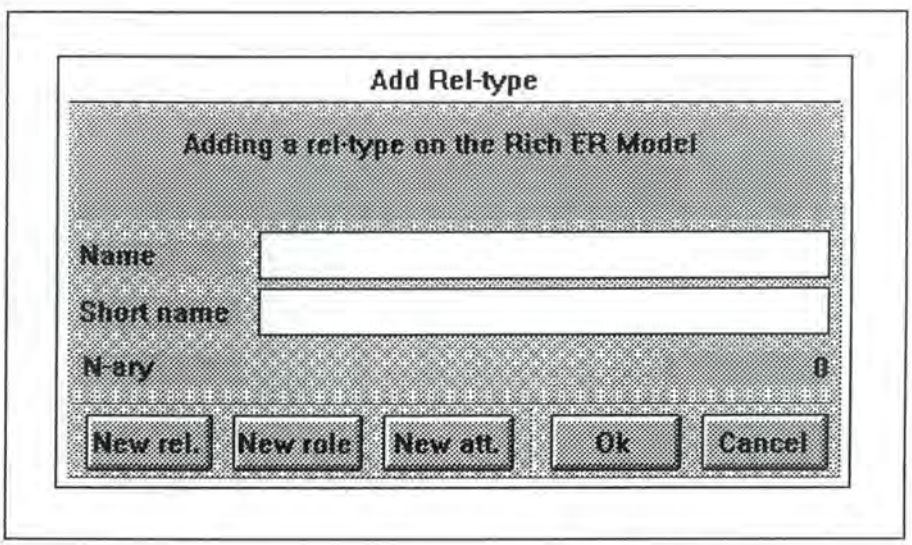


Figure 8 - 1 : The dialog box 'Add Rel-type'

- fill in the name
- activate twice the push button 'New role' in order to create the necessary roles
- validate the dialog box by pushing the 'OK' button
  The modification add_rel-type is now executed and it is mapped to the logical level and to the SQL description. In addition, it has an impact on the data and on the program extracts.

  Note:
  - The push button 'New rel.' has the same effect as the 'OK' button, but regenerates in addition the dialog box 'Add Rel-type'.
  - The output field 'n-ary' indicates the number of roles of the relationship-type.

As we already said, DB-MAIN is conceived to support the conception, the reverse-engineering and the evolution of a database. Sometimes the proposed modifications for the conception and for the evolution may have the same name, but do not have the same effects. In the evolutionary approach, a modification operates not only on the conceptual schema (as in the conception approach), but also on the logical one and on the SQL description. Moreover, it has to take into account the concerned data and program extracts. For example, removing an attribute when conceiving a database is much simpler then removing one on a populated database. In addition to the data problem, populated databases often have application programs referencing it, whereas databases in the conceptual phase do not. Moreover in the evolutionary approach the modifications are only allowed on the conceptual level whereas during the conception the modifications are also possible on the other levels (e.g. on the logical one).

In order to avoid this problem and that one of the possible mix-up of the modifications, we would propose to distinguish three different working modes: the conception, the reverse-engineering and the evolution. Each mode will, of course, include only its own modifications. The user can switch between the three modes by pushing on the buttons situated on the bottom of the screen: the 'C' button stands for conception, the 'R' button for reverse-engineering and the 'E' button for evolution. In the evolution mode, the DB-Main screen would look as in Figure 8-2.

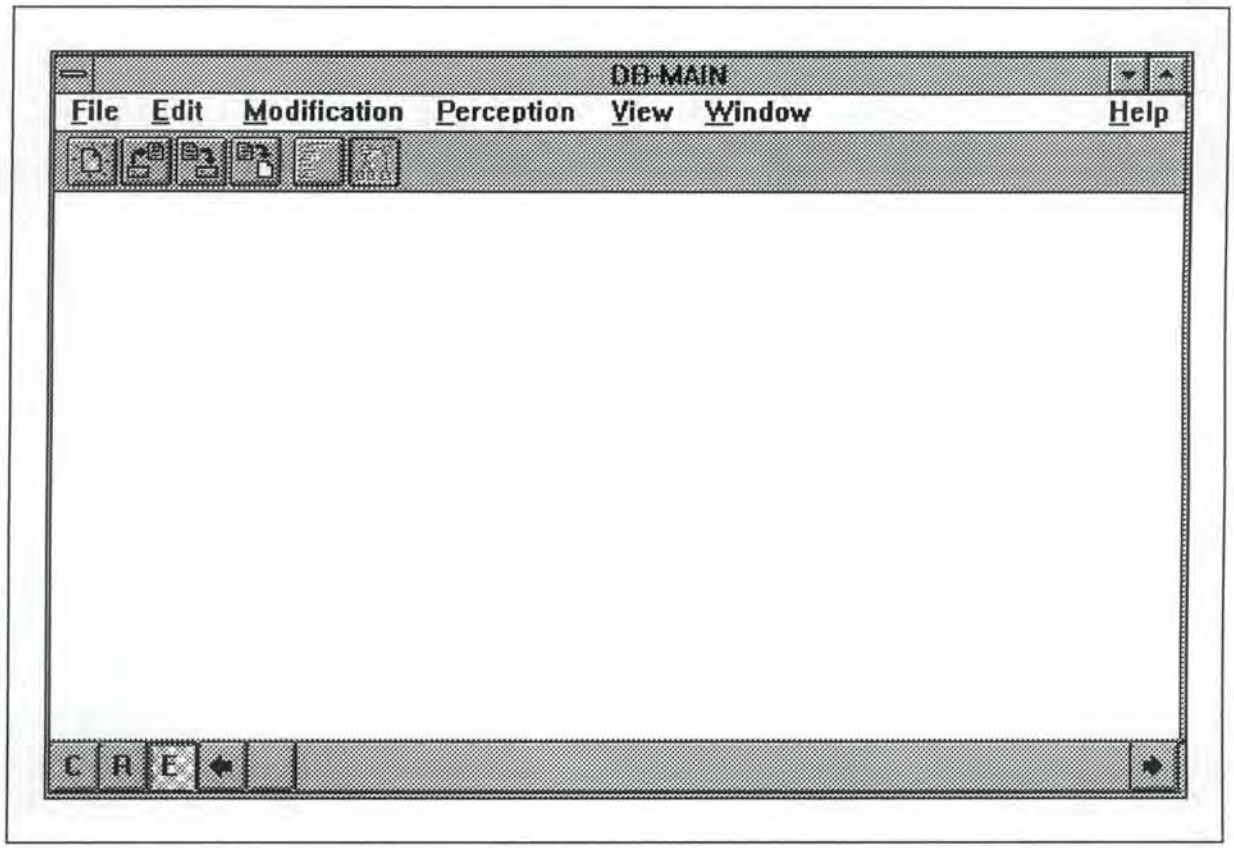


Figure 8 - 2 : The DB-Main screen in the evolution mode

2.2.4. Text Analysis and Name Processing

To address the requirements of the application program analysis, DB-MAIN includes an interactive pattern-matching engine which can search large text files for defined patterns or clichés expressed in PDL, a Pattern Definition Language, which is close to the BNF notation. This engine can be used in our modifications to detect the concerned programs extracts. In order to get an optimal detection the set of patterns can be split into two parts: the generic pattern and the specific one. The generic patterns define, for example, the skeleton of an SQL query in general whereas the specific patterns complete this skeleton by defining it for a precise environment, i.e. a specific database system and host language. If a matching program extract has been found, it can be changed automatically or manually, depending on the modification and/or on the user.

DB-MAIN also includes a name processor with which selected names in a schema or text can be modified according to substitution patterns. This processor is very useful, especially for our rename modifications. Some examples of such substitution patterns are:

- « ^C- » → « CUST- »	replaces each prefix « C- » by the prefix « CUST- »
- « DATE » → « TIME »	replaces each substring « DATE », whatever its position, by the substring « TIME »

In addition, the name processor allows case transformations: lower-to-upper, upper-to-lower, capitalise and remove accents. These parameters can be saved as a name processing script and reused later.

2.2.5. The Assistants

An assistant is a higher-level tool dedicated to solving a special kind of problems , or conducting specific activities. It gives access to the basic toolboxes of DB-MAIN, but in a controlled and intelligent way.

DB-MAIN currently includes two assistants supporting database evolution: the Modification Assistant and the Analysis Assistant.

The Modification Assistant
It allows applying one or several modifications to selected objects. The assistant allows the user to build a script comprising a list of modifications, execute it, save and load it. Predefined scripts are available to modify any schema according to specific ER models. Note that DB-MAIN permits the user to define customized modifications (see part 2.2.6. page 8-7).

The Analysis Assistant
This tool is dedicated to the analysis of schemas. The first step consists in defining a submodel as a restriction of the generic specification model. This restriction appears as a boolean expression of elementary predicates stating which specification patterns are valid (for example: a relationship-type has exactly two roles). A submodel appears as a script which can be saved and loaded. Predefined submodels are available: normalised ER, binary ER and so on. Customized predicates can here also be added. The second step consists then in evaluating the current schema against a specific submodel. This provides a list which describes the detected violations.

2.2.6. Functional Extensibility

No CASE tool can satisfy the needs of all users in any possible situation. DB-MAIN provides a set of built-in standard tools. These tools are sufficient to satisfy most basic needs in database engineering. However, specialised operators may be needed to deal with unforeseen or marginal situations. In addition, analysing and generating texts in any language, or importing and exchanging specifications with any CASE tool is practically impossible, even with highly parametric import/export processors. To cope with such problems, DB-MAIN provides the VOYAGER-2 (V2) development environment allowing users to build their own functions, whatever their complexity. V2 offers a powerful language in which specific DB-MAIN tools can be developed.

2.2.7. Methodological Control and Design Recovery

In the context of database application evolution, understanding how the engineering processes have been carried out when legacy systems have been developed, and guiding today analysts in conducting application development, maintenance and reengineering, are major functions that should be offered by the CASE tools. In DB-MAIN, the design includes thus not only the specifications, but also the reasonings, the modifications, the hypotheses, the decisions which the development process was made of.

2.3. ARCHITECTURE OF THE DB-MAIN TOOL

After having seen the different components of the DB-MAIN tool, we will show how they interact with each other. The architecture of the tool is depicted in Figure 8-3.

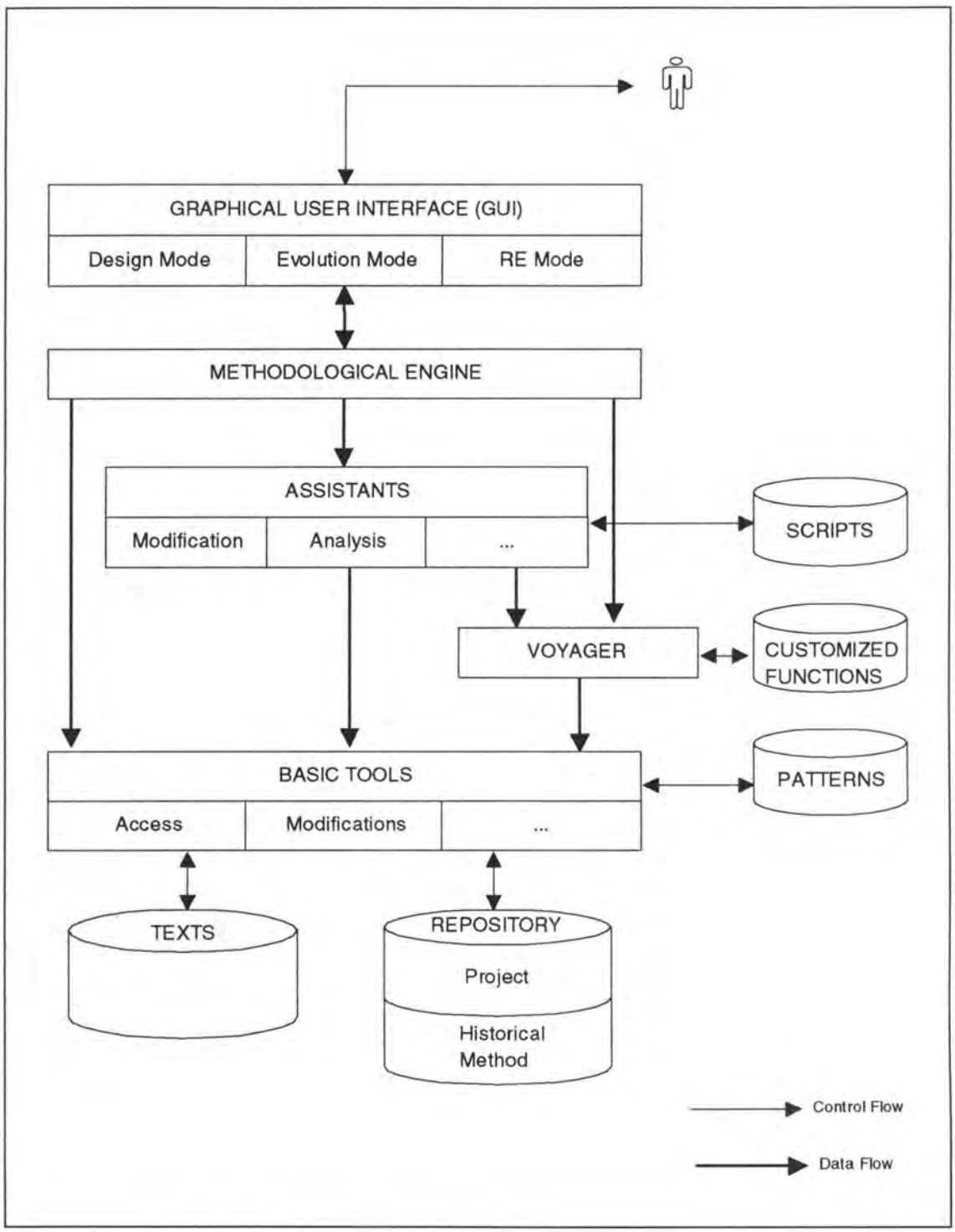


Figure 8 - 3 : The architecture of the DB-MAIN tool

Chapter 9:

Conclusion

In our thesis, we have tried to tackle the database evolution problem by studying schema modifications and their impacts. As we have said in chapter 1, we have situated our thesis in the idealistic context of the forward database maintenance strategy. We have therefore only considered schema modifications on the conceptual level and studied their impacts onto the logical level, the SQL description, the data and the application programs. On the conceptual level, we have chosen the ER model as it is a widely spread model. However, due to the richness of this model, studying the modifications on it would have been very complex and, in a first approach, we have had to simplify it.

We have therefore started by choosing a restricted relational model, the Basic Relational Model and we have tried to build an ER model, the Kernel, whose concepts are entirely translatable into those of the Basic Relational Model. Once these two models were defined, we have started studying the modifications in a very restricted framework. For this study, we have chosen an inductive approach: we have first studied the different modifications on a case study before treating them in general.

As the Kernel is however a very poor ER model, we have enriched it into what we called the Basic ER Model. When mapping the concepts of the Basic ER Model down to the logical level, we have observed that the Basic Relational Model was not sufficient anymore and we have thus used the Rich Relational Model instead. This enrichment forced us not only to review the previously considered modifications, but also to introduce new modifications. Due to time constraints however, we could only give indications about them by illustrating them on examples.

We have then underlined that the Basic ER Model is not powerful enough to represent the whole ER model. We have therefore had to introduce the Rich ER Model, which allows the most commonly used concepts and we have had to study how the modifications are mapped down to the Basic ER Model. As these modifications strongly interfere with those of the Basic ER Model and as we have had no time to study the latter ones, we have been unable to study them in detail.

Since our thesis is only useful in the context of a CASE tool supporting database evolution, we have briefly discussed how to integrate our work into a specific CASE tool, namely the DB-MAIN tool.

Due to the complexity of the schema modifications of the ER model, we have not been able to tackle the whole problem. Indeed, we have only studied the modifications on a restricted model and we would therefore propose a second thesis which should try to analyse them on the complete ER model. This extension should be based upon the work that we have realised. Note that a lot of work has still to be done and that the result is far from being evident as some modifications will require contradictory constraints or limitations. Once the complete study accomplished, the work could be integrated into a CASE tool.

Our thesis together with the proposed extension can also be seen as a contribution to a multi-view system supporting database evolution. We studied such a multi-view system during our practical training in Australia. It is an environment which allows a user to switch between several views of a database schema: on the conceptual level, the NIAM and the ER/EER views and on the logical level, the relational one. Such a system is represented in Figure 9-1.

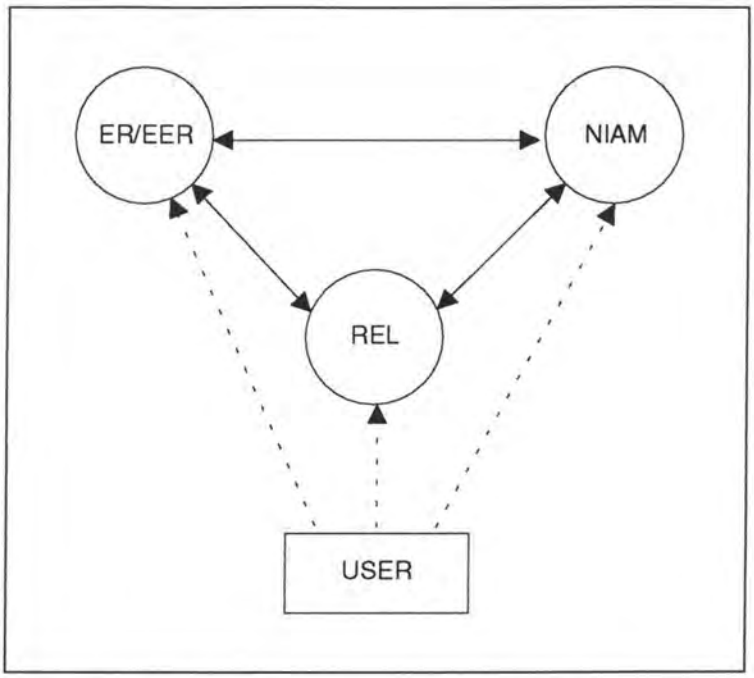


Figure 9 - 1 : A multi-view system supporting database evolution

In this system, any modification made on a schema in one view must directly be propagated, by one or several modifications, to the corresponding schema in the other views. Note that the schemas represented in the different views must therefore be semantically equivalent. Our thesis together with its extension would only have solved the problem of the mapping of the modifications of the ER model down to the relational one. Further developments would then have to tackle the remaining mapping problems among which the mapping of the modifications of the relational model up to the ER model. Note however that there is still a long way to go in order to obtain such a multi-view system.

Bibliography

[ARI91] *Temporally oriented data definitions: Managing schema evolution in temporally oriented DB*
G. Ariav
Data & Knowledge Engineering 6 (1991) -- pages 451-467

[BOD89] *Conception assisté des systèmes d'information - Méthode/Modèles/Outils - 2ième edition*
F. Bodart and Y. Pigneur
Masson (1989)

[BEL93] *An Active Meta-model for Knowledge Evolution in an Object-oriented Database 1993*
Z. Bellahsene
in Proceedings of CAiSE'93, Springer Verlag (1993) -- pages 39-53

[CAS93] *On the design and maintenance of optimized relational representations of entity-relationship schemas*
M.A. Casanova, L. Tucherman and A.H.F. Laender
Data & Knowledge Engineering 11 (1993) --pages 1-20

[DAT89] *A Guide to The SQL Standard - Second Edition*
C.J. Date
Addison-Wesley Publishing Company (1989)

[DAT95] *An Introduction to Database Systems Sixth Edition*
C.J. Date
Addison-Wesley Publishing Company (1995)

[DAT86] *Relational Database - Selected Writings*
C.J. Date
Addison-Wesley Publishing Company (1986)

[DBM94] *DB-MAIN Tutorial*
Version 2 - November 1994

[DB289] *IBM Database 2 Version 2 - SQL Reference*
Release 2
IBM (September 1989)

[ELM94] *Fundamentals Of Database Systems - Second Edition*
R. Elmasri and S.B. Navathe
The Benjamin/Cummings Publishing Company, Inc. (1994)

[EWA93] *A Procedural Approach to Schema Evolution*
C.A. Ewald and M.E. Orlowska
in Proceedings of CAiSE'93, Springer Verlag (1993) -- pages 22-38

[HAI88] *Introduction à la théorie relationnelle des bases de données*
J-L. Hainaut
FUNDP (mars 1988)

[HAI92] *Introduction à SQL, un système de gestion de bases de données relationnelles*
J.-L. Hainaut
Quatrième version, FUNDP (4 novembre 1992)

[HAI94a] *Database Evolution: the DB-MAIN Approach*
J-L. Hainaut, V. Englebert, J. Henrard, J-M. Hick, D. Roland
FUNDP (1994)

[HAI94b] *The DB-MAIN Database Engineering CASE tool - Functions Overview*
J-L. Hainaut, V. Englebert, J. Henrard, J-M. Hick, D. Roland
FUNDP (July 1994)

[HAI95a] *Requirements for Information System Reverse Engineering Support*
J-L. Hainaut, V. Englebert, J. Henrard, J-M. Hick, D. Roland
FUNDP (24/01/1995)

[HAI95b] *DB-MAIN: un atelier d'ingénierie de bases de données*
J-L. Hainaut, V. Englebert, J. Henrard, J-M. Hick, D. Roland
FUNDP (1995)

[LIU94] *Database Schema Evolution using EVER Diagrams*
C-T. Liu, S-K. Chang and P.K. Chrysanthis
Proc. of Workshop on AVI'94 (1994) -- pages 122-132

[NGU89] *Schema evolution in object-oriented database sytems*
G.T. Nguyen and D. Rieu
Data & Knowledge Engineering 4 (1989) --pages 43-67

[PRO94] *EVORM: A conceptual modelling technique for evolving application domains*
H.A. Proper and T.P. van der Weide
Data & Knowledge Engineering 12 (1994) --pages 313-359

[PRO95] *Information disclosure in evolving information systems: taking a shot at a moving target*
H.A. Proper and T.P. van der Weide
Data & Knowledge Engineering 15 (1995) --pages 135-168

[RDB91] *VAX Rdb/VMS Version 4.1. - SQL Reference Manual*
Digital (December 1991)

[ROD92] *Schema Evolution in Database Systems - An Annotated Bibliography*
J.F. Roddick
SIGMOND RECORD, Vol 21, No 4 (December 1992) -- pages 35-40

[ROD93] *A Taxonomy for Schema Versioning Based on the Relational and Entity Relationship Models*
J.F. Roddick, N.G. Craske and T.J. Richards
Proc. of the 12th Int. Conf. on Er Approach, Arlington Dallas, ER Institute (1993)

[ROD94] *Schema Evolution in Database Systems - An Updated Bibliography*
John F. Roddick
University of South Australia (1994)

[TSU91] *MORE: An Object-Oriented Data Model with a Facility for Changing Object Structures*
K. Tsuda, K. Yamamoto, M. Hirakawa, M. Tanaka and T. Ichikawa
IEEE Transactions on Knowledge and Data Engineering, Vol 3, No 4, (December 1991) -- pages 444-460

[VAN89] *Introduction to SQL - Second Edition*
Rick F. van der Lans
Addison-Wesley Publishing Company (1989)

Appendix 1:

Study of the Modifications: Case Study Approach

TABLE OF CONTENTS

Appendix 1 :Study of the Modifications: General Approach

TABLE OF FIGURES

Appendix 1 :Study of the Modifications: General Approach

1. INTRODUCTION

We have to study in this appendix all the modifications of the conceptual level and their impact on the logical level, on the SQL database structure, on the data and on the application programs.

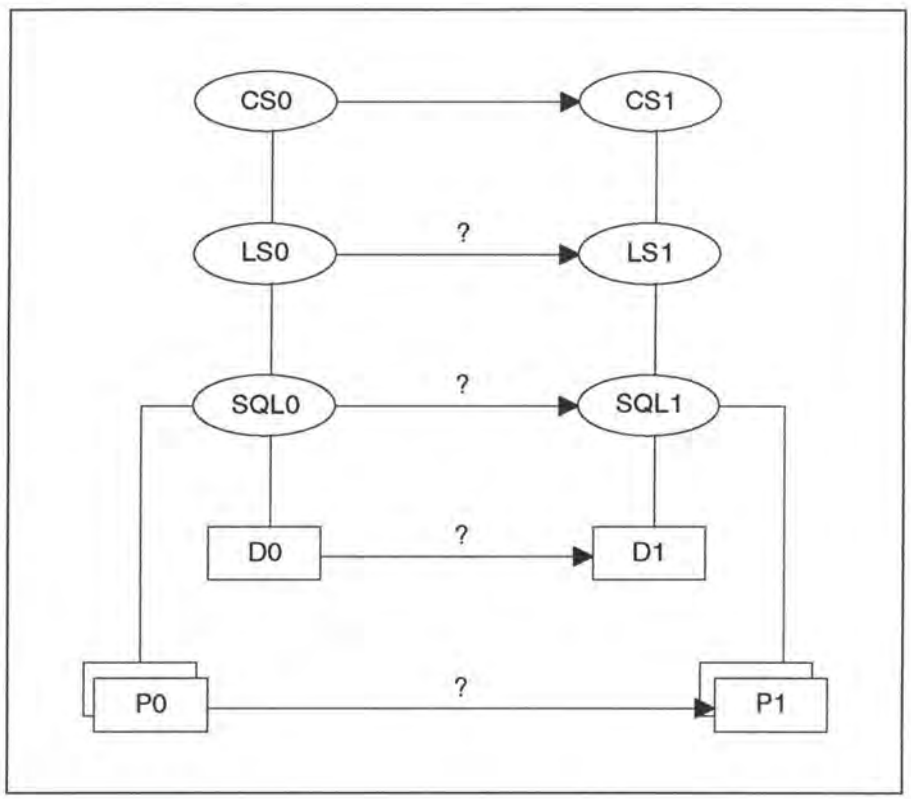


Figure A1 - 1 : Representation of the database evolution problem

If the conceptual schema CS0 has been changed, the logical schema LS0 and the SQL description SQL0 must be changed accordingly. Data D0 is no longer valid and has to be converted into data D1. Finally the applications P0 must be partly rewritten in order to comply with the new data structures described in SQL1.[HAI94a]

As shown in the third chapter, the modifications are classified according to the objects on which they apply on one hand and, on the other hand, according to their nature: augmenting, decreasing or preserving semantics (see page 3-4). In order not to loose the overview of this appendix, we will give once more the typology of the modifications. We will indicate in bold those modifications which are redundant with those detailed in chapter 4.

Modifications of the entity-types which:

augment the semantics:	add_entity-type
decrease the semantics:	remove_entity-type
preserve the semantics:	**rename_entity-type**

Modifications of the relationship-types which:

augment the semantics:	add_1-1/0-1_rel-type add_0-1/0-1_rel-type add_1-1/0-N_rel-type add_0-1/0-N_rel-type
decrease the semantics:	remove_1-1/0-1_rel-type **remove_0-1/0-1_rel-type** remove_1-1/0-N_rel-type remove_0-1/0-N_rel-type
preserve the semantics:	rename_1-1/0-1_rel-type rename_0-1/0-1_rel-type rename_1-1/0-N_rel-type rename_0-1/0-N_rel-type

Modifications of the roles which:

augment the semantics:	**augment_max_card** decrease_min_card
decrease the semantics:	decrease_max_card augment_min_card

Modifications of the attributes which:

augment the semantics:	add_optional_attribute add_mandatory_attribute make_attr_optional extend_domain_attribute change_type_int_char change_type_float_char change_type_date_char change_type_date_int change_type_int_float change_type_date_float

decrease the semantics:	remove_optional_attribute remove_mandatory_attribute **make_attr_mandatory** restrict_domain_attribute change_type_char_int change_type_float_int change_type_char_float change_type_char_date change_type_int_date change_type_float_date
preserve the semantics:	rename_optional_attribute rename_mandatory_attribute

Modifications of the identifiers which:

augment the semantics:	remove_unique_feature
decrease the semantics:	add_unique_feature
preserve the semantics:	**switch_PK_unique**

For each object, we will thus distinguish three types of modifications: those augmenting, decreasing and preserving the semantics. Within each of these three parts, we will develop for each modification its impact on the Logical Schema, on the SQL Description & Data and on the Program Extracts.

2. MODIFICATIONS OF THE ENTITY-TYPES

2.1. MODIFICATIONS WHICH AUGMENT THE SEMANTICS

2.1.1. Add_entity-type[1]

Note:
Each entity-type must have at least one attribute and must have a primary key.

Let us suppose we want to add entity-type FACTORY to our case study example:

[1]Normally we would have to add the following precondition: 'The name of the entity-type that should be added must not yet exist'. As such preconditions are trivial, we will not indicate them anymore.

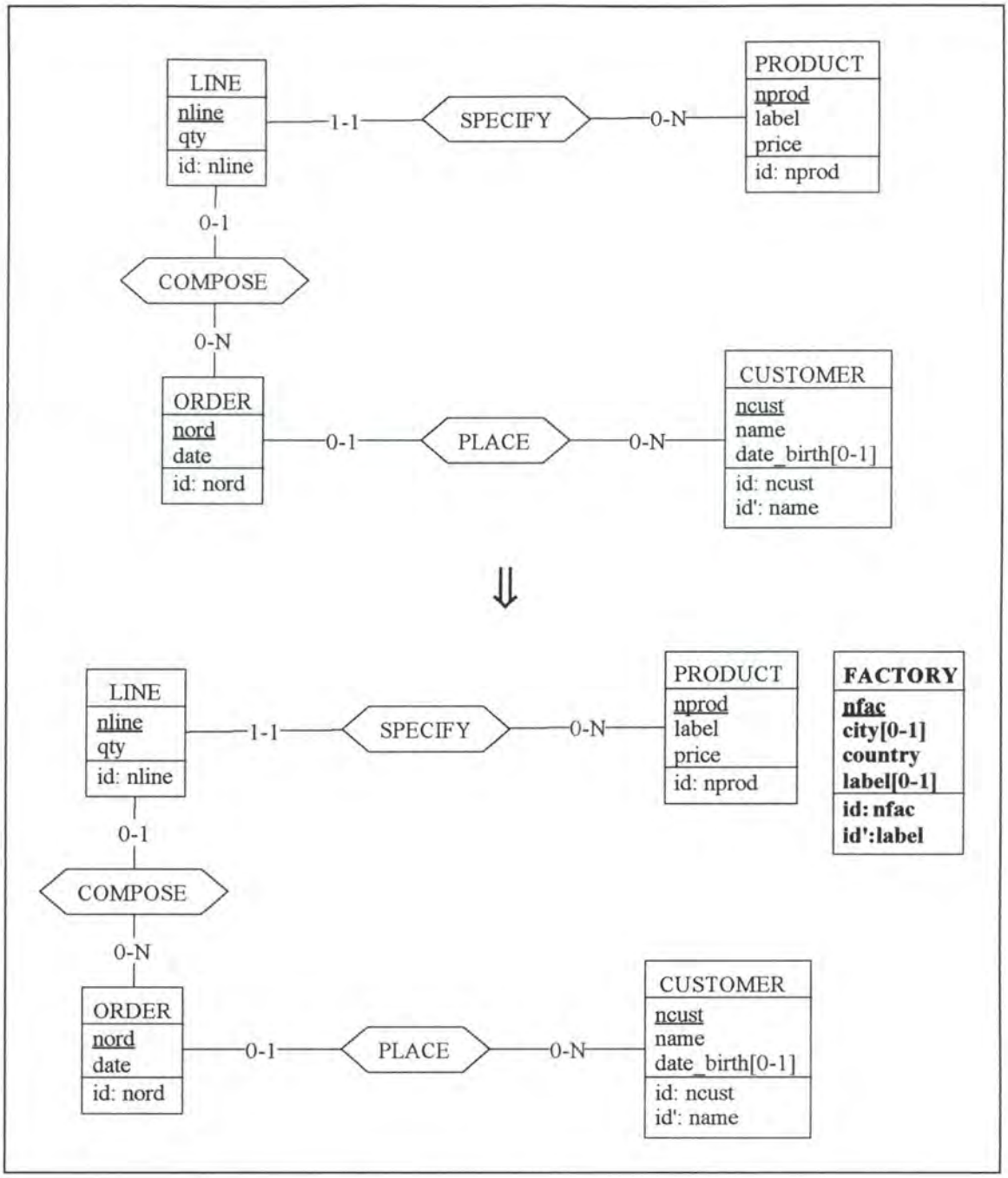


Figure A1 - 2 : Adding an entity-type on the conceptual level

2.1.1.1. Logical Schema

On the logical level, we have to add the corresponding relation to the schema.

2.1.1.2. SQL Description & Data

```
create table FACTORY
      ( nfac            smallint  not null      constraint F_nfac,
        city            char(20),
```

```
country          char(30)  not null      constraint F_country,
label            char(20),
primary key (nfac) constraint idFAC1,
unique (label) constraint idFAC2 )
```

There is no effect on the existing data as we only add a new table.

2.1.1.3. Program Extracts

There is no change on the existing application programs. The documentation must however be updated. As the changes of the documentation are necessary for all the modifications, we will not indicate them anymore in this appendix.

2.2. MODIFICATIONS WHICH DECREASE SEMANTICS

2.2.1. Remove_entity-type

Precondition:
The entity-type that has to be removed must not be connected to any relationship-type.

Let us suppose we want to remove the entity-type FACTORY.

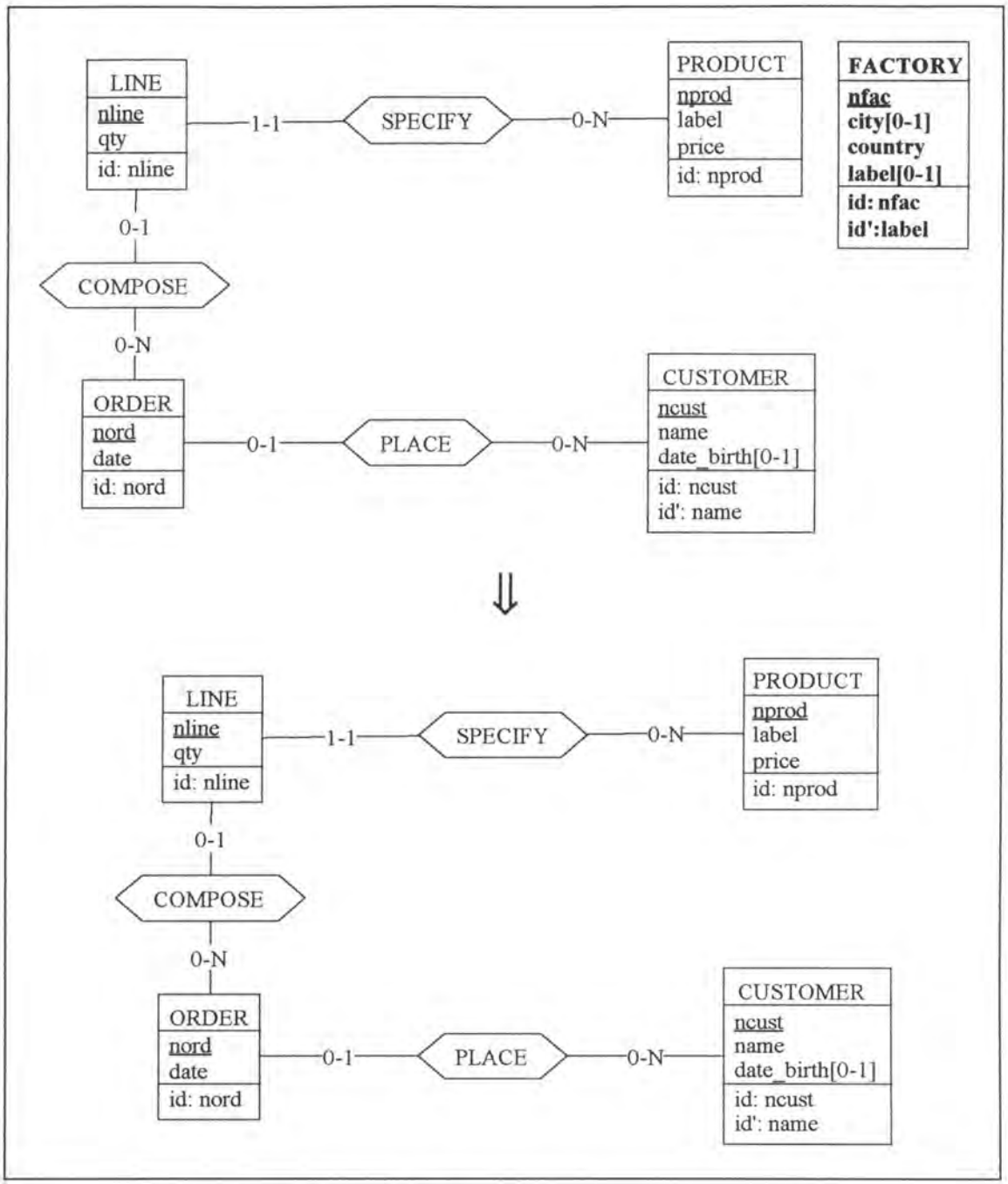


Figure A1 - 3 : Removing an entity-type on the conceptual level

2.2.1.1. Logical Schema

In the logical schema we remove the relation FACTORY.

2.2.1.2. SQL Description & Data

```
drop table FACTORY cascade;
```

Note that all the data included in table FACTORY will be lost.

2.2.1.3. Program Extracts

The select queries which reference table FACTORY are invalid. The application programs in which they appear must thus be reviewed. We cannot describe a general method how to deal with these application programs as each one of them must be treated individually, depending on its context. A CASE tool offering this modification should only indicate the concerned program extracts and give hints about possible changes or removals.

2.3. MODIFICATIONS WHICH PRESERVE THE SEMANTICS

2.3.1. Rename_entity-type

Let us suppose we want to change in our case study the entity-type CUSTOMER into CLIENT.

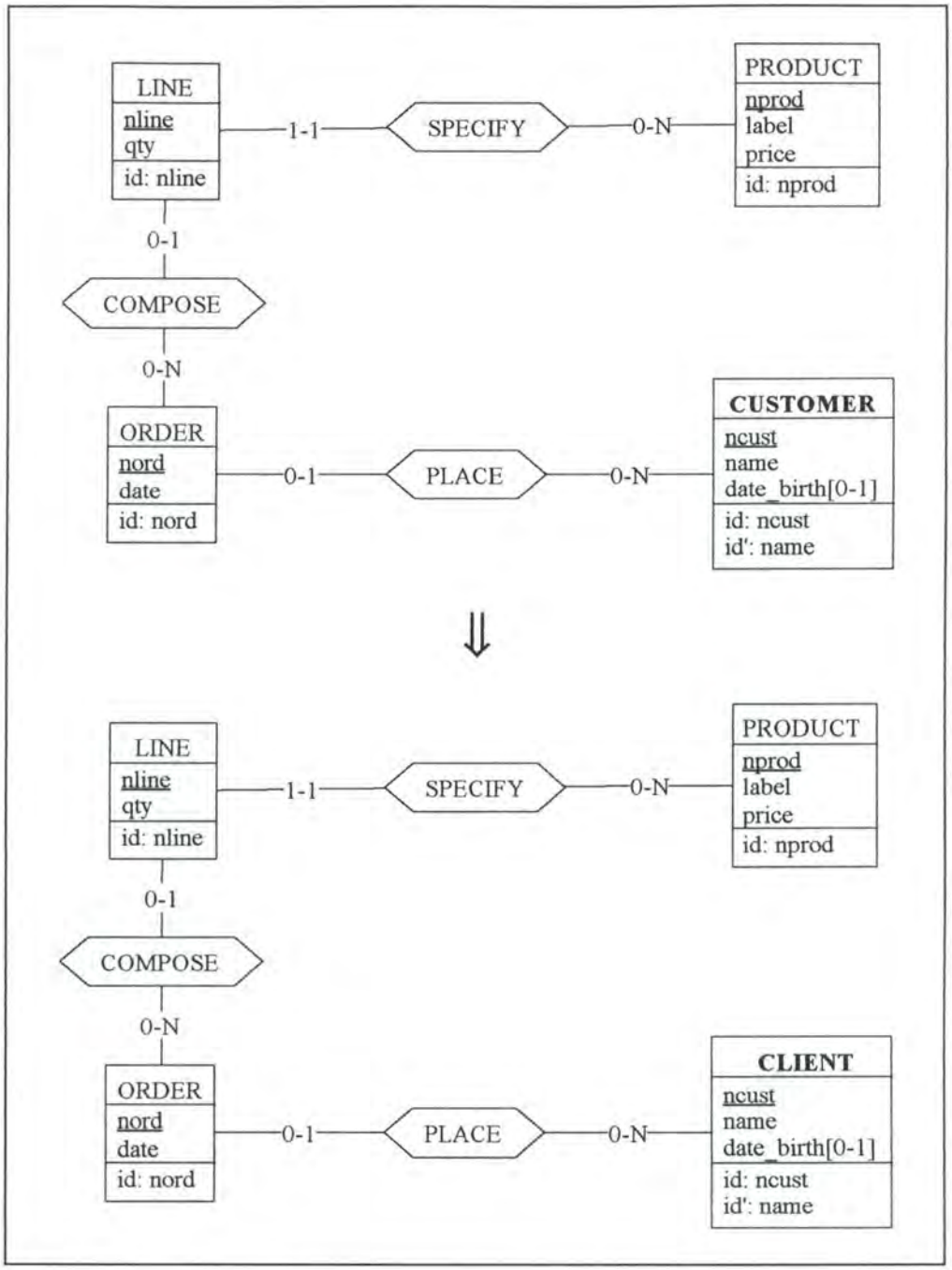


Figure A1 - 4 : Renaming an entity-type on the conceptual level

2.3.1.1. Logical Schema

In the logical schema, we have to change the name of the corresponding relation.

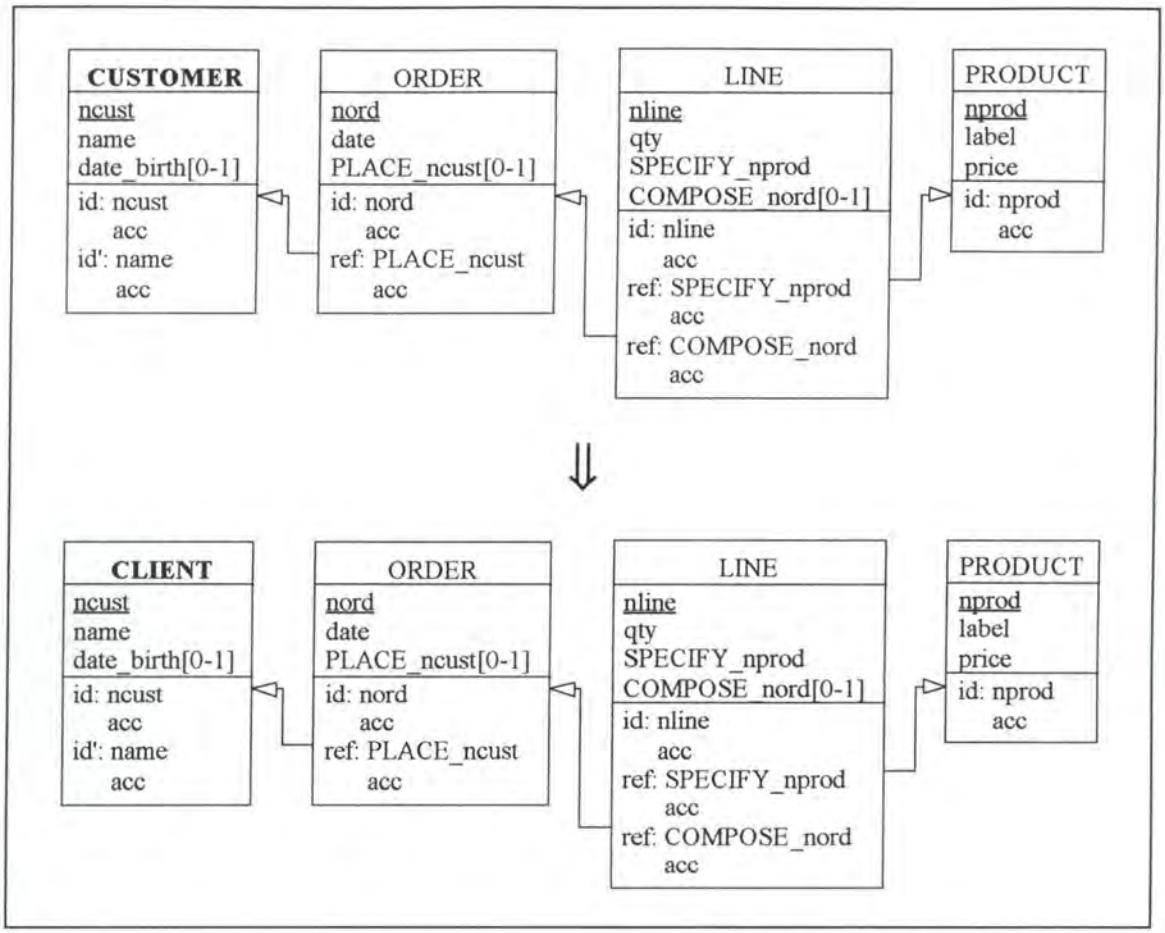


Figure A1 - 5 : Renaming an entity-type on the logical level

2.3.1.2. SQL Description & Data

In some SQL languages there may be a 'rename table' command. The modification would thus become:

```
alter table CUSTOMER
  rename table CLIENT on cascade;
```

In SQL-RDB however, no such command exists and we have therefore to create a new table and to copy the data into it.

```
create table CLIENT
        ( ncust          char(4)     not null    constraint C_ncust,
          name           char(12)    not null    constraint C_name,
          date_birth     date,
          primary key (ncust) constraint idCLI1,
          unique (name)       constraint idCLI2 );
insert into CLIENT
  select ncust, name, date_birth
  from CUSTOMER;
alter table ORDER
  drop constraint CUS1,
  add constraint foreign key (PLACE_ncust) references CLIENT
                                                  constraint CLI1;
```

```
(* For each view defined on table CUSTOMER, we have to redefine it on table
CLIENT. In future we will not consider views anymore as they do not
correspond to ER objects. *)

drop table CUSTOMER cascade;
```

No data is lost as the data is just moved from one table into another.

Notes:

- This operation in SQL-RDB is often a very slow one as we have to copy a whole table. We thus recommend to create a view CLIENT which includes only the table CUSTOMER. This could be realised by the following command:

```
create view CLIENT
  as select *
     from CUSTOMER
```

- Other SQL languages, such as DB2, offer another possibility to implement the modification: giving a synonym to the entity-type (that has to be renamed) instead of renaming it properly. This alternative could be realised by the following SQL command:

```
create synonym CLIENT
  for CUSTOMER;
```

Note that in both cases the original table is however not renamed.

2.3.1.3. Program Extracts

In all the select queries referencing CUSTOMER, we have to rename it with CLIENT. For example, the first select query of our case study (see page 4-7) would become:

```
select *
  from CLIENT
  where date_birth = 09/06/1969
```

In addition to the select queries, we have also to review the application programs in which they appear. For instance, we must rename certain variables and/or some fields or headings in the user interfaces. Finally, let us note that the documentation should also be updated.

3. MODIFICATIONS OF THE RELATIONSHIP-TYPES

3.1. MODIFICATIONS WHICH AUGMENT THE SEMANTICS

3.1.1. Add_1-1/0-1_rel-type

Let us suppose we have an entity-type ADDRESS having as attributes nadd, street, number, zip and city, and having as primary key nadd. We want now to link ADDRESS to the entity-type CUSTOMER of our case study by a 1-1/0-1 relationship-type LIVE.

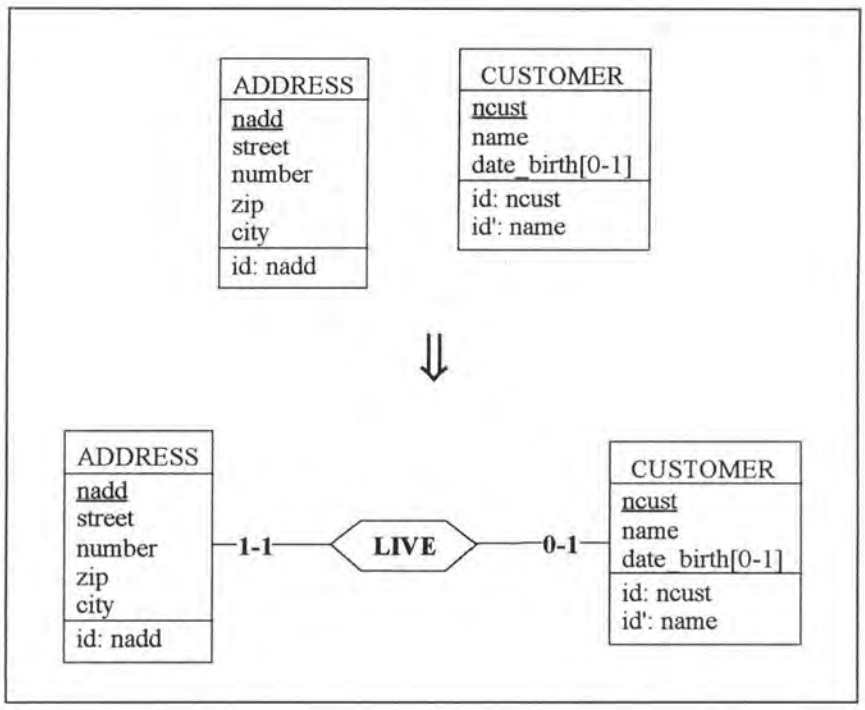


Figure A1 - 6 : Adding a 1-1/0-1 relationship-type on the conceptual level

3.1.1.1. Logical Schema

In the logical schema we add the primary key ncust of CUSTOMER to ADDRESS as a foreign and a candidate key.

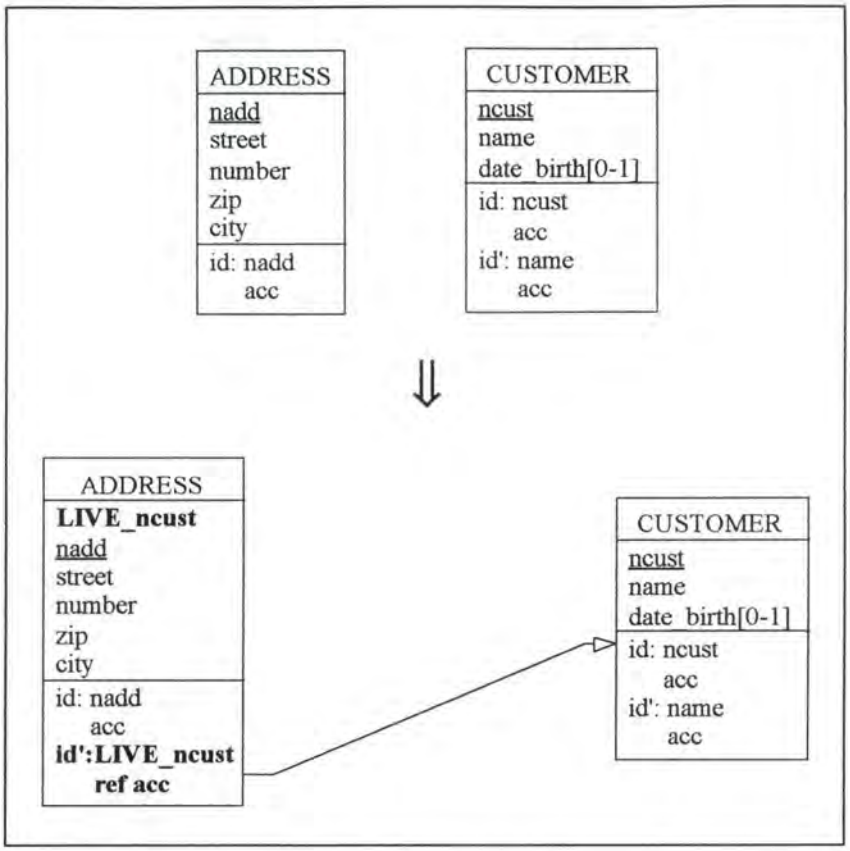


Figure A1 - 7 : Adding a 1-1/0-1 relationship-type on the logical level

3.1.1.2. SQL Description & Data

```
alter table ADDRESS
  add LIVE_ncust char(4) default '0000' not null constraint A_LIVE_ncust;

(* The user has to introduce the data into column LIVE_ncust representing
   the relationship-type LIVE. He must be aware that the rows of ADDRESS
   which have no data specified for column LIVE_ncust will be deleted. *)

delete from ADDRESS
  where LIVE_ncust = '0000';
alter table ADDRESS
  add constraint unique (LIVE_ncust) constraint idADD2,
  add constraint foreign key (LIVE_ncust) references CUSTOMER
                                                  constraint CUS1;
```

3.1.1.3. Program Extracts

Note:
Sometimes select queries taken out of their program environment are not sufficient to study completely the impact on the application programs, as they do not show, for instance, the changes that must be made on the variables. We therefore consider in some cases embedded queries.

Let us consider the following program extract:

```
var nadd:   INTEGER;
    street: STRING[20];
    number: INTEGER;
    zip:    INTEGER;
    city:   STRING[20];

   :
exec SQL
      declare c cursor for
        select *
          from ADDRESS
          where city = 'Adelaide';
            :
      open c;
      fetch c into :nadd, :street, :number, :zip, :city;
end exec;
while SQLCODE = 0       (* the last item has not yet been treated *)
  do begin
       write (nadd);
       write (street);
       write (number);
       write (zip);
       write (city);
       exec SQL
            fetch c into :nadd, :street, :number, :zip, :city;
       end exec
     end;
exec SQL
      close c
end exec;
   :
```

To adapt this part of program to the changes made on table ADDRESS, we propose two potential modifications:

- We modify the embedded SQL to:

```
        :
declare c cursor for
  select nadd, street, number, zip, city
    from ADDRESS
      :
```

- We add a variable customer corresponding to the new column LIVE_ncust:

```
var customer: STRING[4];
    nadd: ...
        :

   :
exec SQL
      declare c cursor for
        select *
          from ADDRESS
          where city = 'Adelaide';
            :
      open c;
      fetch c into :nadd, :street, :number, :zip, :city, :customer;
end exec;
while SQLCODE = 0       (* the last item has not yet been treated *)
  do begin
       write (nadd);
       write (street);
```

```
        write (number);
        write (zip);
        write (city);
        write (customer);
        exec SQL
             fetch c into :nadd, :street, :number, :zip, :city,
                                                        :customer;
        end exec
      end;
exec SQL
      close c
end exec;
   :
```

The user interfaces may also be changed: for example, a CUSTOMER is now displayed with its living ADDRESS.

3.1.2. Add_0-1/0-1_rel-type

Let us suppose we have an entity-type ADDRESS having as attributes nadd, street, number, zip and city and having as primary key nadd. We want now to link ADDRESS to the entity-type CUSTOMER of our case study by a 0-1/0-1 relationship-type WORK.

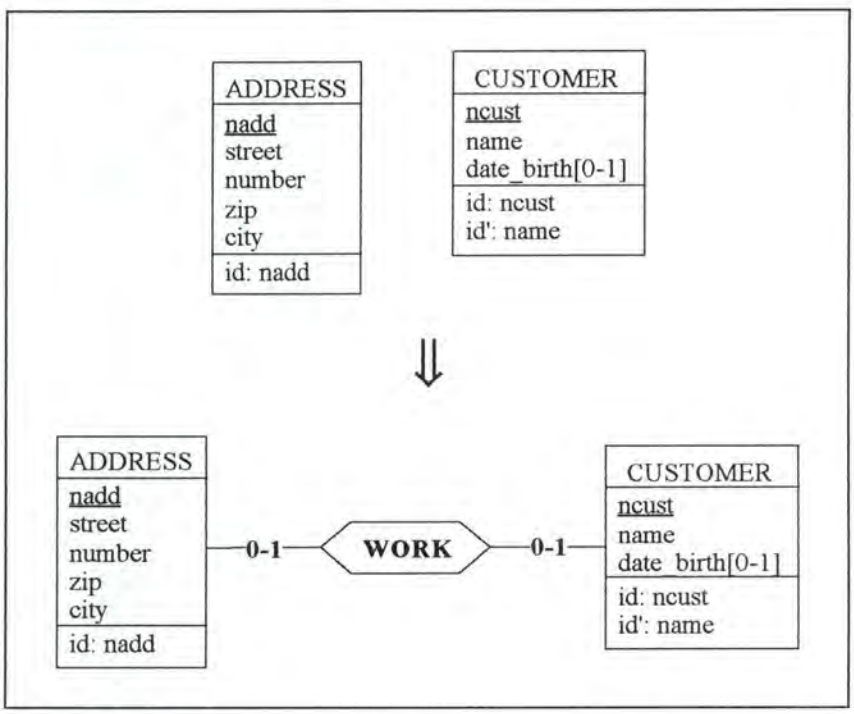


Figure A1 - 8 : Adding a 0-1/0-1 relationship-type on the conceptual level

There are two possible representations on the logical level for the relationship-type WORK:

- WORK is implemented by a foreign key in ADDRESS
- WORK is implemented by a foreign key in CUSTOMER

3.1.2.1. WORK is implemented by a foreign key in ADDRESS

3.1.2.1.1. Logical Schema

In the logical schema, we add the primary key ncust of CUSTOMER to ADDRESS as an optional foreign and candidate key.

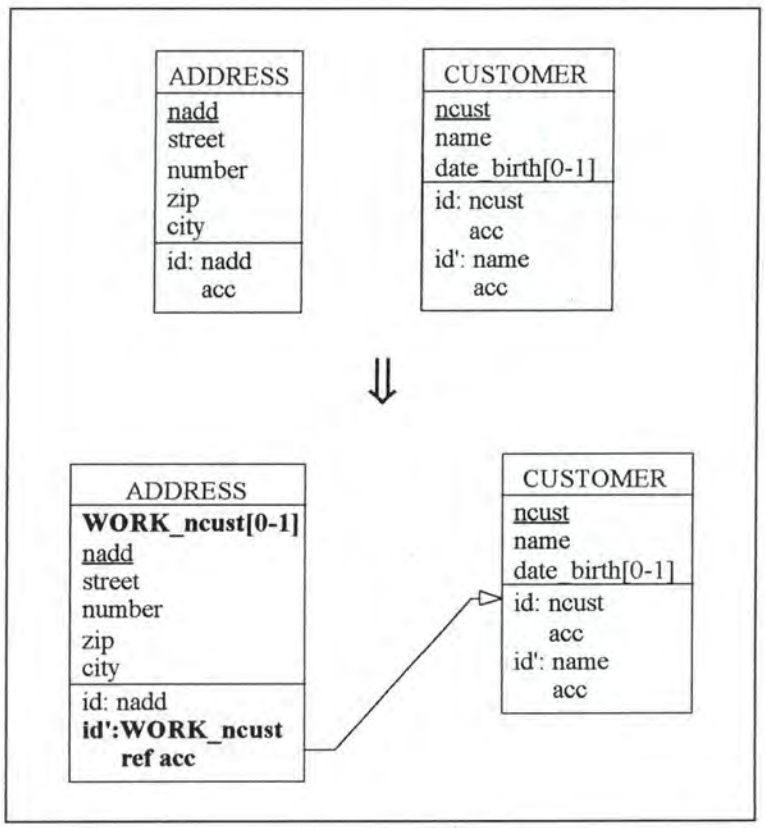


Figure A1 - 9: Adding a 0-1/0-1 relationship-type on the logical level

3.1.2.1.2. SQL Description & Data

```
alter table ADDRESS
  add WORK_ncust    char(4),
  add constraint unique (WORK_ncust)  constraint idADD2,
  add constraint foreign key (WORK_ncust) references CUSTOMER
                                                  constraint CUS1;
```

Note that all the rows of ADDRESS have a null value for column WORK_ncust.

3.1.2.1.3. Program Extracts

We are confronted with the same problem as in the case add_1-1/0-1_rel-type (see page A1-13).

3.1.2.2. WORK is implemented by a foreign key in CUSTOMER

This case is symmetrical to the previous one (see page A1-16).

3.1.3. Add_1-1/0-N_rel-type

Let us suppose that we have an entity-type FACTORY having as primary key nfac and the entity-type PRODUCT of our case study. So far there is no relationship-type between PRODUCT and FACTORY. We want now to add a 1-1/0-N relationship-type MANUFACTURE between these two entity-types.

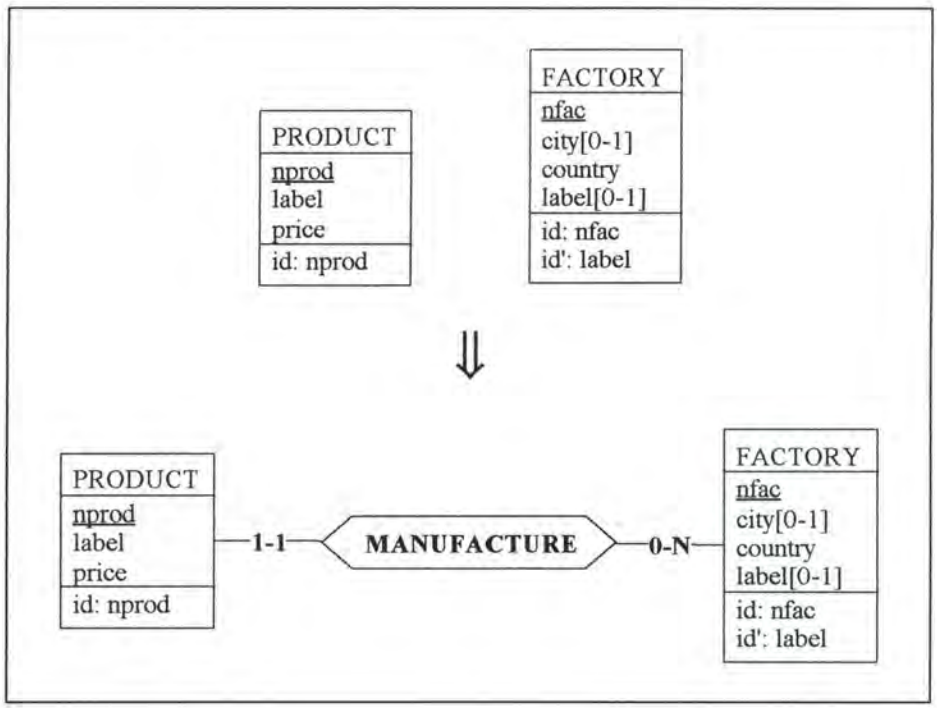


Figure A1 - 10 : Adding a 1-1/0-N relationship-type on the conceptual level

3.1.3.1. Logical Schema

We add the primary key nfac of FACTORY to PRODUCT as a mandatory foreign key.

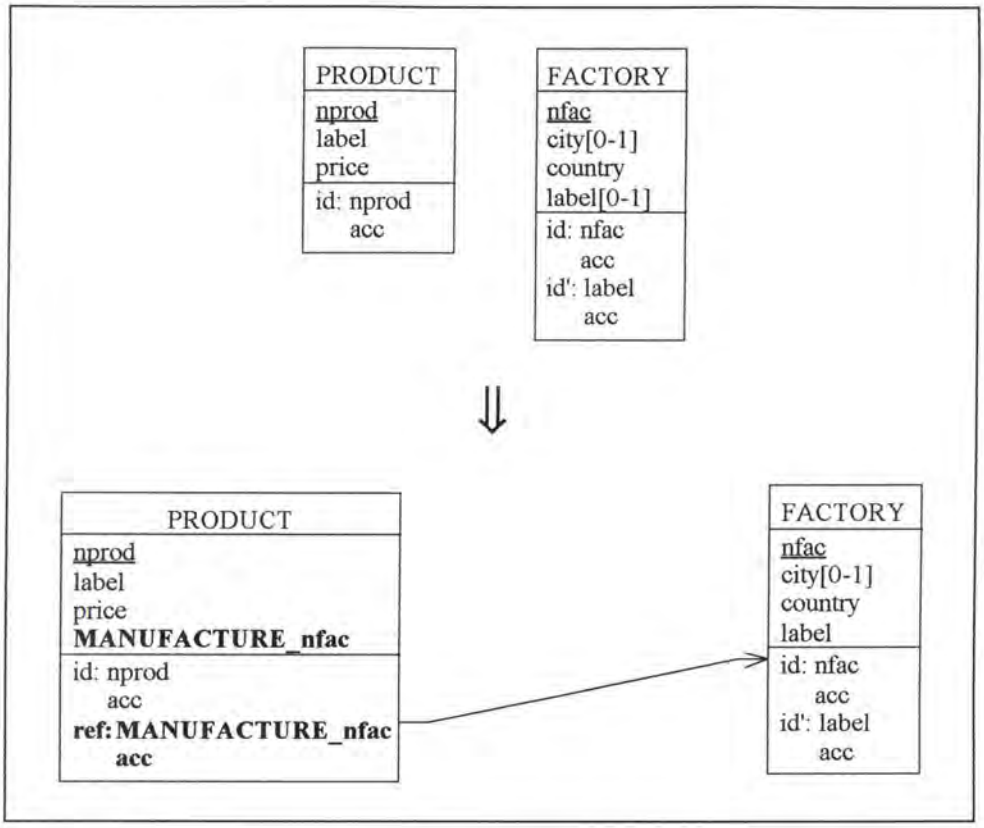


Figure A1 - 11 : Adding a 1-1/0-N relationship-type on the logical level

3.1.3.2. SQL Description & Data

```
alter table PRODUCT
  add MANUFACTURE_nfac  smallint  default 0  not null
                                        constraint P_MANUFACTURE_nfac;

(* The user has to introduce the data into column MANUFACTURE_nfac
representing the relationship-type MANUFACTURE. He must be aware that the
rows of PRODUCT which have no data specified for column MANUFACTURE_nfac
will be deleted because of the foreign key constraint. *)

delete from PRODUCT
  where MANUFACTURE_nfac = 0;
alter table PRODUCT
  add constraint foreign key (MANUFACTURE_nfac) references FACTORY
                                                    constraint FAC1;
```

3.1.3.3. Program Extracts

The modifications on the application programs are similar to those of the case add_1-1/0-1_rel-type (see page A1-13).

3.1.4. Add_0-1/0-N_rel-type

Let us suppose that we have again the entity-type FACTORY having as primary key nfac and the entity-type PRODUCT of our case study. So far there is no relationship-type between these entity-types. We want this time to add a 0-1/0-N relationship-type MANUFACTURE between them.

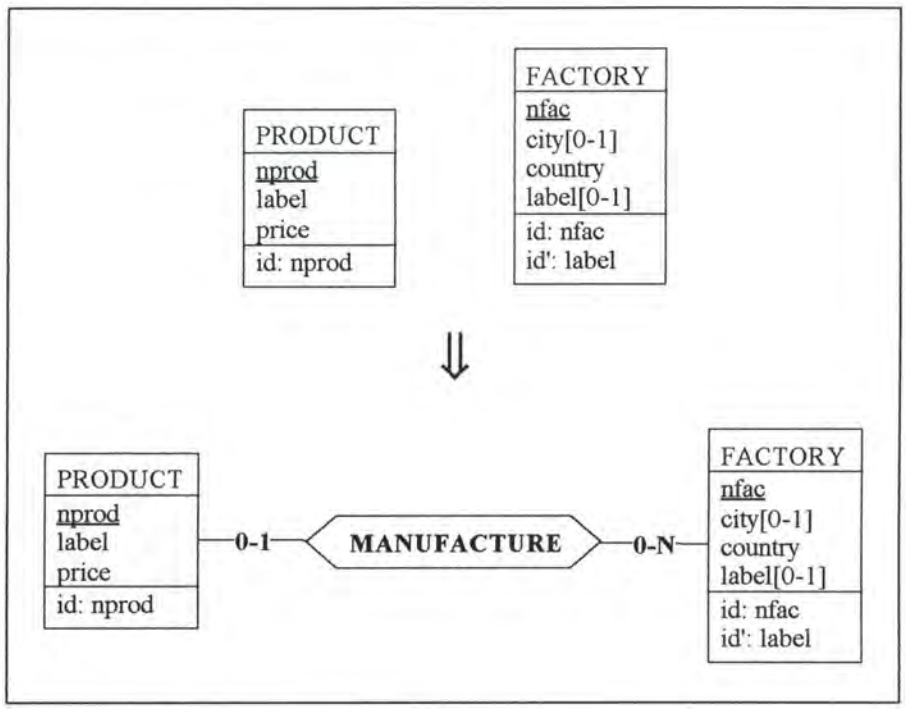


Figure A1 - 12 : Adding a 0-1/0-N relationship-type on the conceptual level

3.1.4.1. Logical Schema

We add the primary key nfac of FACTORY to PRODUCT as an optional foreign key.

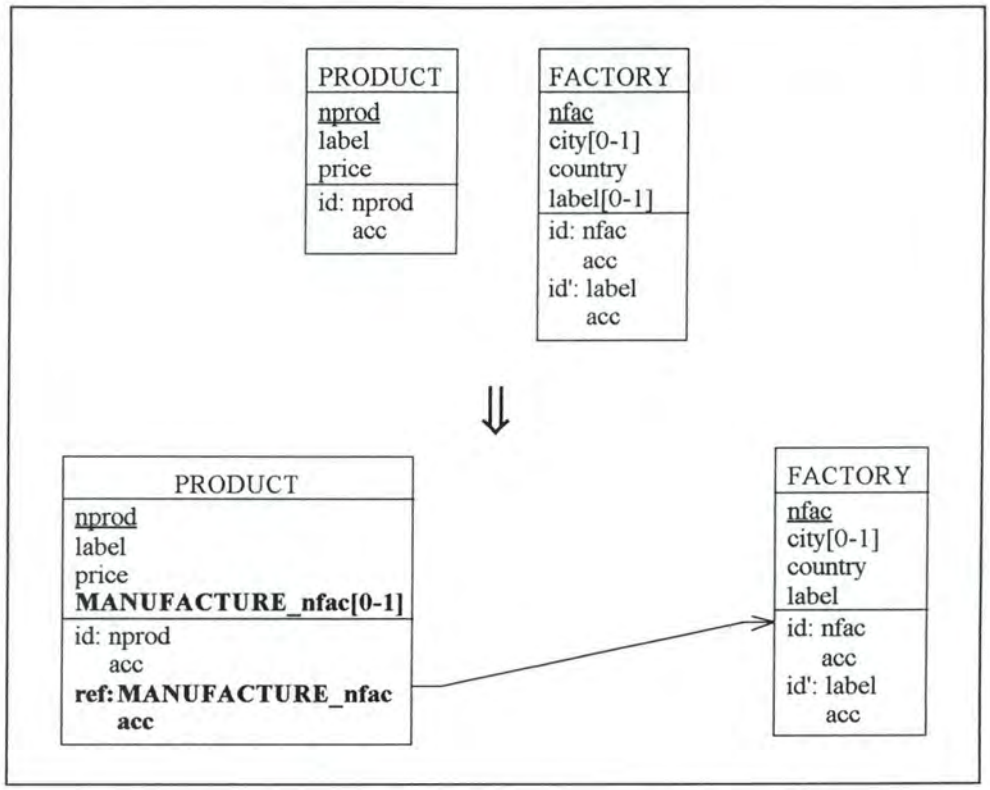


Figure A1 - 13 : Adding a 0-1/0-N relationship-type on the logical level

3.1.4.2. SQL Description & Data

```
alter table PRODUCT
  add MANUFACTURE_nfac  smallint,
  add constraint foreign key (MANUFACTURE_nfac) references FACTORY
                                                    constraint FAC1;
```

Note that all the rows of table PRODUCT have a null value for column MANUFACTURE_nfac, as shown in Figure A1-14.

PRODUCT			
nprod	label	price	(MANUFACTURE_nfac)
AA110	christmas tree	35	null
CA510	glass	50	null
AB099	pencil	10	null
BE072	gearbox	1000	null
WN592	wheel	850	null
SW226	alarm-clock	75	null
LS906	poster	60	null
SG953	toothbrush	13	null
:	:	:	:

PRODUCT.MANUFACTURE_nfac in FACTORY.nfac

Figure A1 - 14 : The resulting table PRODUCT

3.1.4.3. Program Extracts

The modifications on the application programs are similar to those of the case add_1-1/0-1_rel-type (see page A1-13).

3.2. MODIFICATIONS WHICH DECREASE THE SEMANTICS

3.2.1. Remove_1-1/0-1_rel-type

Let us suppose that we want to remove the 1-1/0-1 relationship-type LIVE.

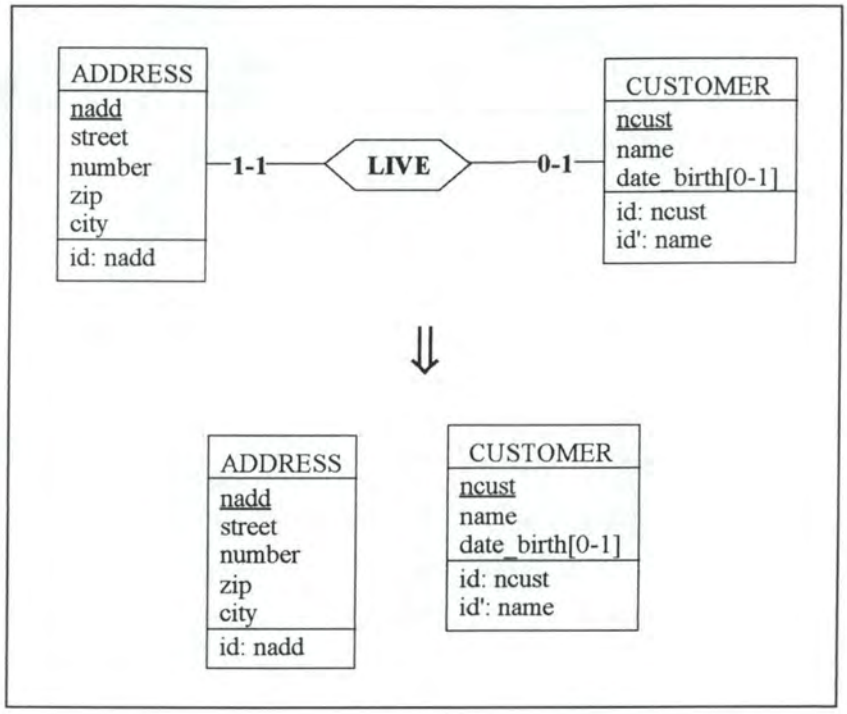


Figure A1 - 15 : Removing a 1-1/0-1 relationship-type on the conceptual level

3.2.1.1. Logical Schema

We remove the column LIVE_ncust in ADDRESS with its candidate and foreign key features.

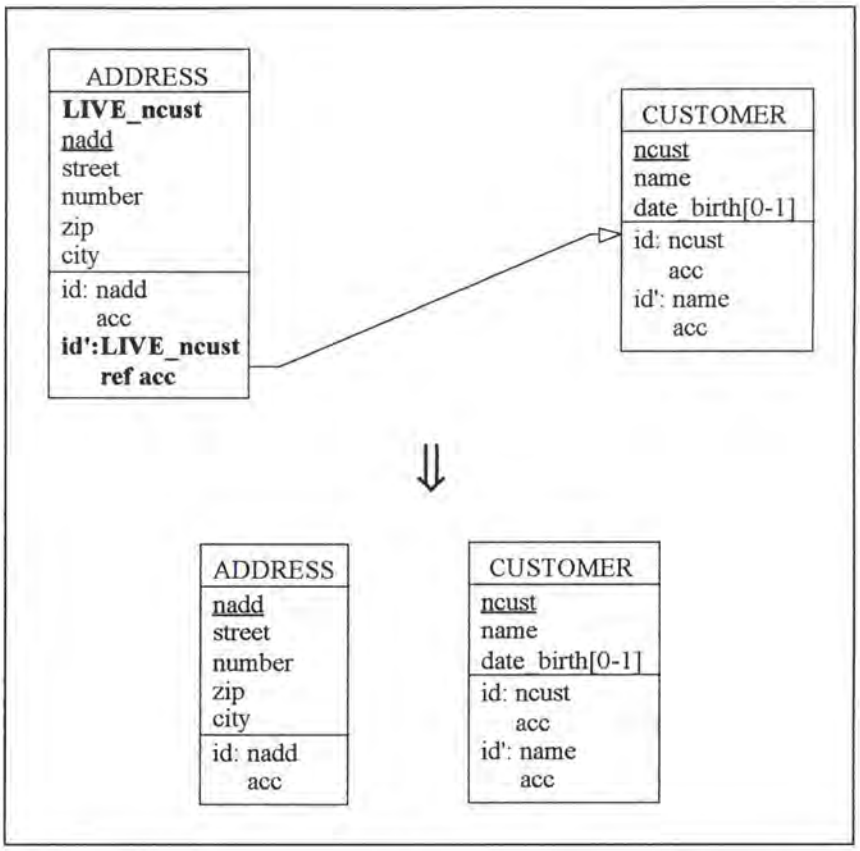


Figure A1 - 16 : Removing a 1-1/0-1 relationship-type on the logical level

3.2.1.2. SQL Description & Data

```
alter table ADDRESS
  drop constraint idADD2,      (* we remove the unique key feature *)
  drop constraint CUS1,        (* we remove the foreign key feature *)
  drop constraint A_LIVE_ncust,     (* we remove the mandatory feature from
                                       column LIVE_ncust *)
  drop LIVE_ncust;
```

The link between a CUSTOMER and his/her ADDRESS where he/she lives is lost.

3.2.1.3. Program Extracts

All the select queries which reference LIVE_ncust in ADDRESS must be modified. For example:

```
select name, street, number, zip, city
  from ADDRESS, CUSTOMER
  where (LIVE_ncust = ncust) and
        (ncust in (select PLACE_ncust
                   from ORDER
                   where nord in (select COMPOSE_nord
                                  from LINE
                                  where SPECIFY_nprod = 'AA110')))
```

⇓

```
select name
  from CUSTOMER
  where ncust in ( select PLACE_ncust
                   from ORDER
                   where nord in ( select COMPOSE_nord
                                   from LINE
                                   where SPECIFY_nprod = 'AA110' ))
```

The application programs in which these queries appear must also be reviewed. We cannot describe a general method how to deal with these application programs as each one of them must be treated individually. Let us for example consider a screen which displays the information about a CUSTOMER, including his/her living ADDRESS. As we have now lost the link between a CUSTOMER and his/her living ADDRESS, the user has to decide what should happen to the part of the screen allocated to the living ADDRESS. He can either drop it and rearrange the screen or reuse it for another purpose (for example: for indicating the working ADDRESS of the CUSTOMER). In addition, the user has to check whether the variables are still all needed.

3.2.2. Remove_0-1/0-1_rel-type

Let us suppose that we want to remove the 0-1/0-1 relationship-type WORK between ADDRESS and CUSTOMER.

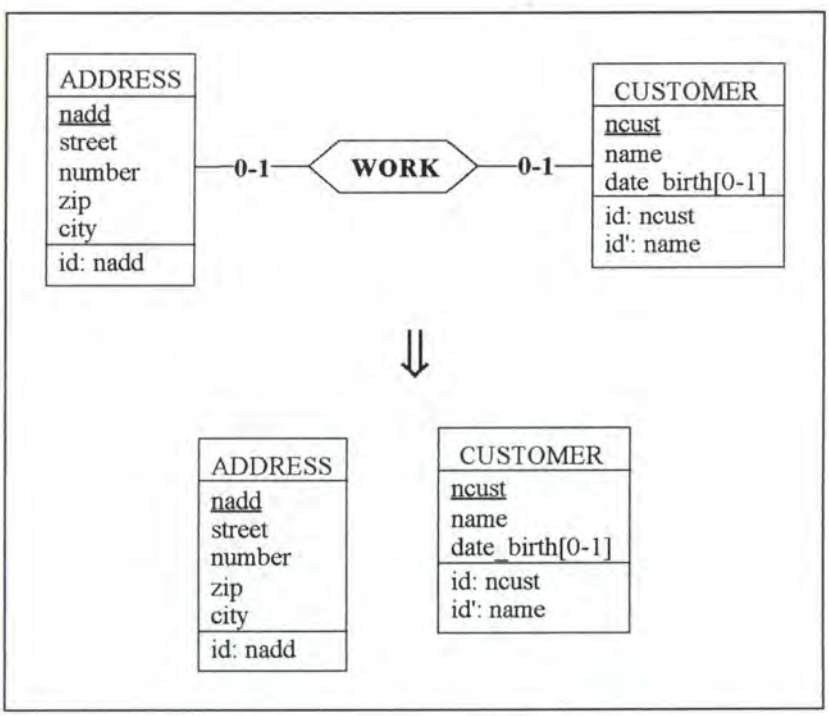


Figure A1 - 17 : Removing a 0-1/0-1 relationship-type on the conceptual level

We have to reconsider the two possible implementations for the relationship-type WORK:

- WORK is implemented by a foreign key in ADDRESS
- WORK is implemented by a foreign key in CUSTOMER

3.2.2.1. WORK is implemented by a foreign key in ADDRESS

3.2.2.1.1. Logical Schema

We remove the column WORK_ncust from ADDRESS with its candidate and foreign key features.

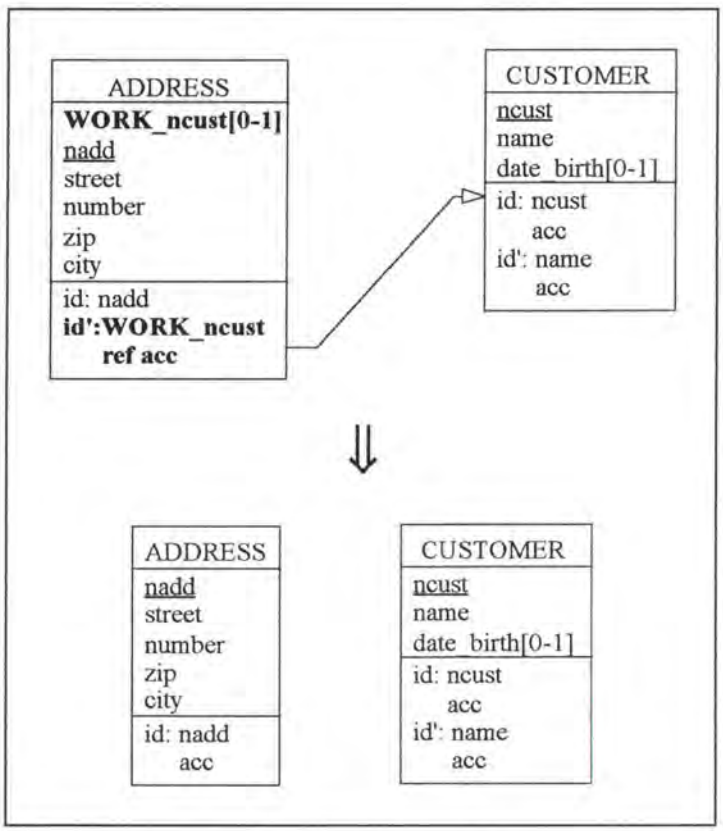


Figure A1 - 18 : Removing a 0-1/0-1 relationship-type on the logical level when it is implemented by a foreign key in table ADDRESS

3.2.2.1.2. SQL Description & Data

```
alter table ADDRESS
  drop constraint idADD2,      (* we remove the unique key feature *)
  drop constraint CUS1,        (* we remove the foreign key feature *)
  drop WORK_ncust;
```

The link between a CUSTOMER and his/her ADDRESS is lost.

3.2.2.1.3. Program Extracts

The impacts on the application programs are similar to those of the case remove_1-1/0-1_rel-type (see page A1-23).

3.2.2.2. WORK is implemented by a foreign key in CUSTOMER

3.2.2.2.1. Logical Schema

We remove the column WORK_nadd in CUSTOMER with its candidate and foreign key features.

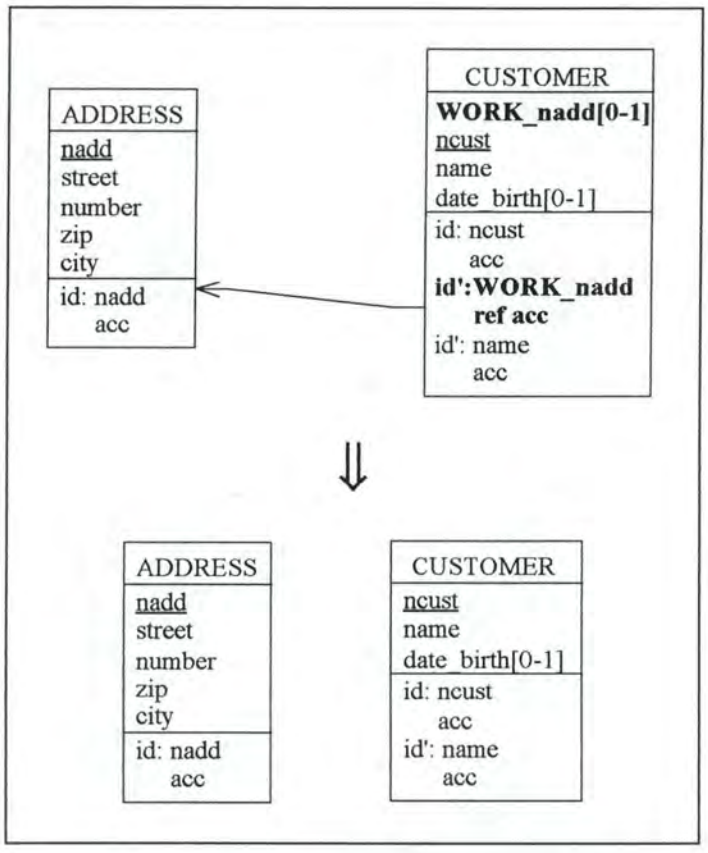


Figure A1 - 19 : Removing a 0-1/0-1 relationship-type on the logical level

3.2.2.2.2. SQL Description & Data

```
alter table CUSTOMER
  drop constraint idCUS2,      (* we remove the unique key feature *)
  drop constraint ADD1,        (* we remove the foreign key feature *)
  drop WORK_nadd;
```

The link between a CUSTOMER and his/her ADDRESS is lost.

3.2.2.2.3. Program Extracts

The impacts on the application programs are similar to those of the modification remove_1-1/0-1_rel-type (see page A1-23).

3.2.3. Remove_1-1/0-N_rel-type

Let us remove the 1-1/0-N relationship-type SPECIFY between LINE and PRODUCT from our case study example.

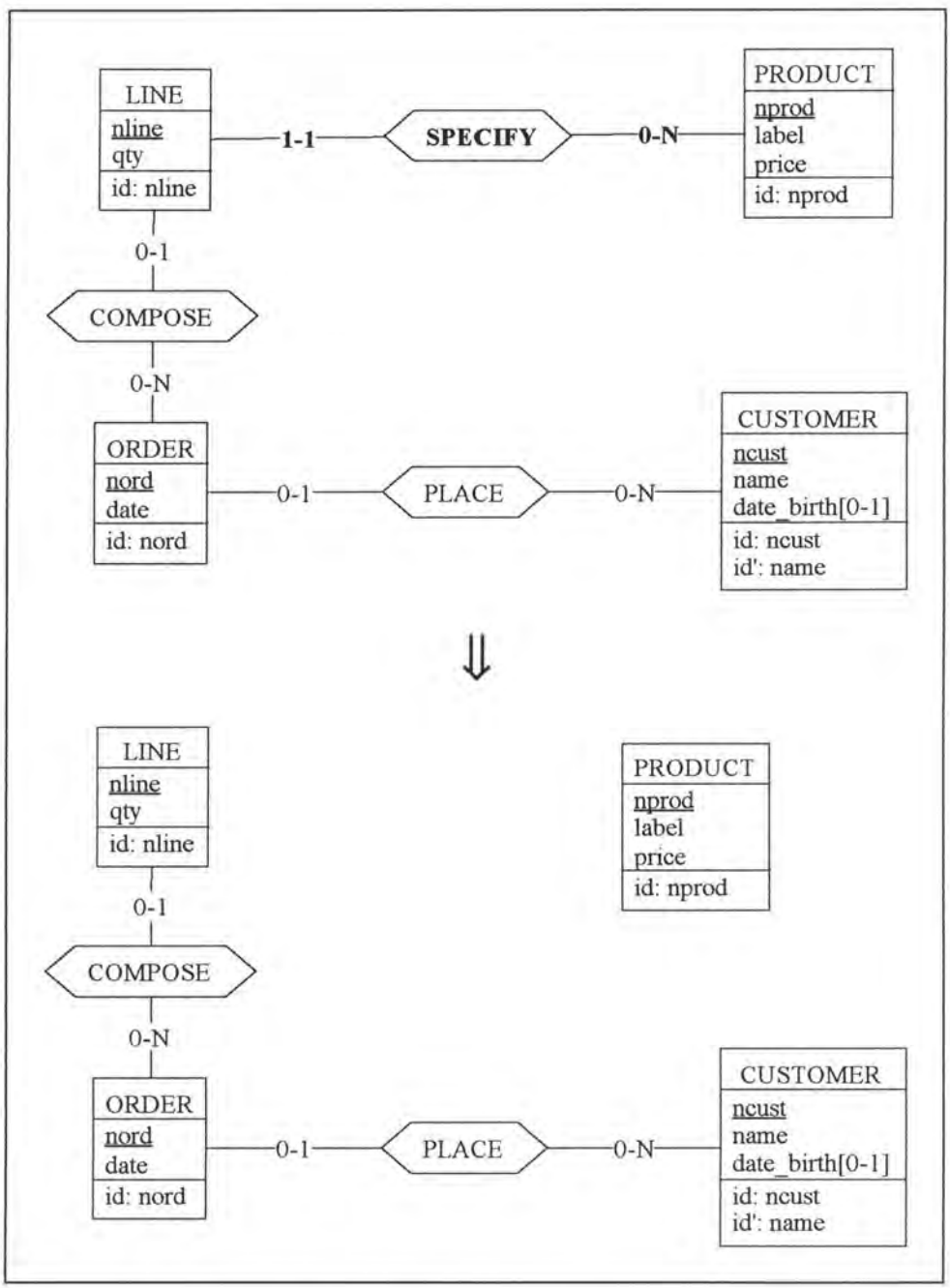


Figure A1 - 20 : Removing a 1-1/0-N relationship-type on the conceptual level

3.2.3.1. Logical Schema

We remove the column SPECIFY_nprod in LINE with its foreign key feature.

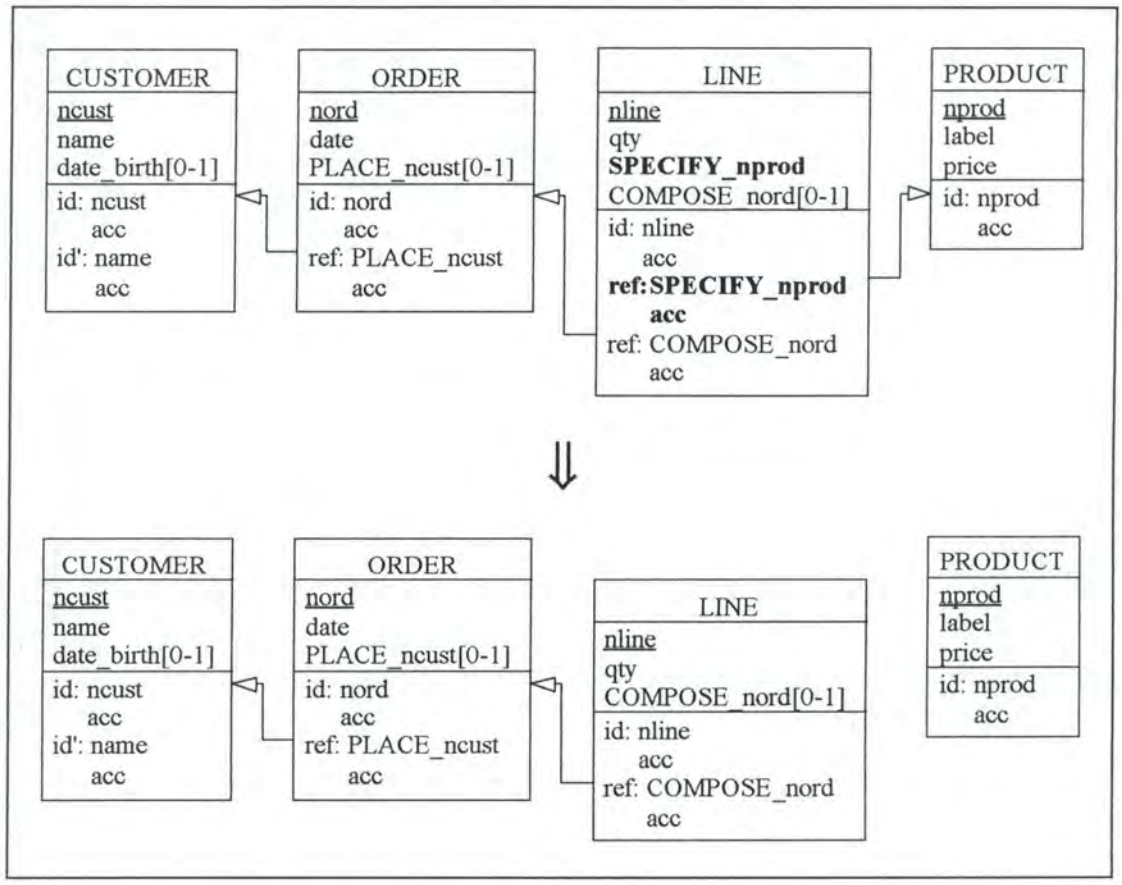


Figure A1 - 21 : Removing a 1-1/0-N relationship-type on the logical level

3.2.3.2. SQL Description & Data

```
alter table LINE
  drop constraint PRO1, (* we remove the foreign key feature *)
  drop constraint L_SPECIFY_nprod,  (* we remove the mandatory feature from
                                       column SPECIFY_nprod *)
  drop SPECIFY_nprod;
```

The link between a LINE and the PRODUCT it specifies is lost as shown in Figure A1-22.

LINE		
nline	(COMPOSE_nord)	qty
AB1234	E386	1000
GH2345	null	1518
RT3456	F285	345
ZU4567	G274	2536
ER5678	null	4587
NM6789	G274	5558
:	:	:

LINE.COMPOSE_nord in ORDER.nord

Figure A1 - 22 : Table LINE when the link to table PRODUCT is lost

3.2.3.3. Program Extracts

The second SELECT and the PROJECT queries of our case study (see page 4-7) must be dropped as they do not make sense anymore. Moreover, the UNION query (see page 4-8) may be modified as follows:

```
select nprod
  from PRODUCT
  where price < = 50.
```

Note that the application programs must be reviewed in a similar way as for the case remove_1-1/0-1_rel-type (see page A1-23).

3.2.4. Remove_0-1/0-N_rel-type

Let us remove the 0-1/0-N relationship-type PLACE between ORDER and CUSTOMER from our case study example.

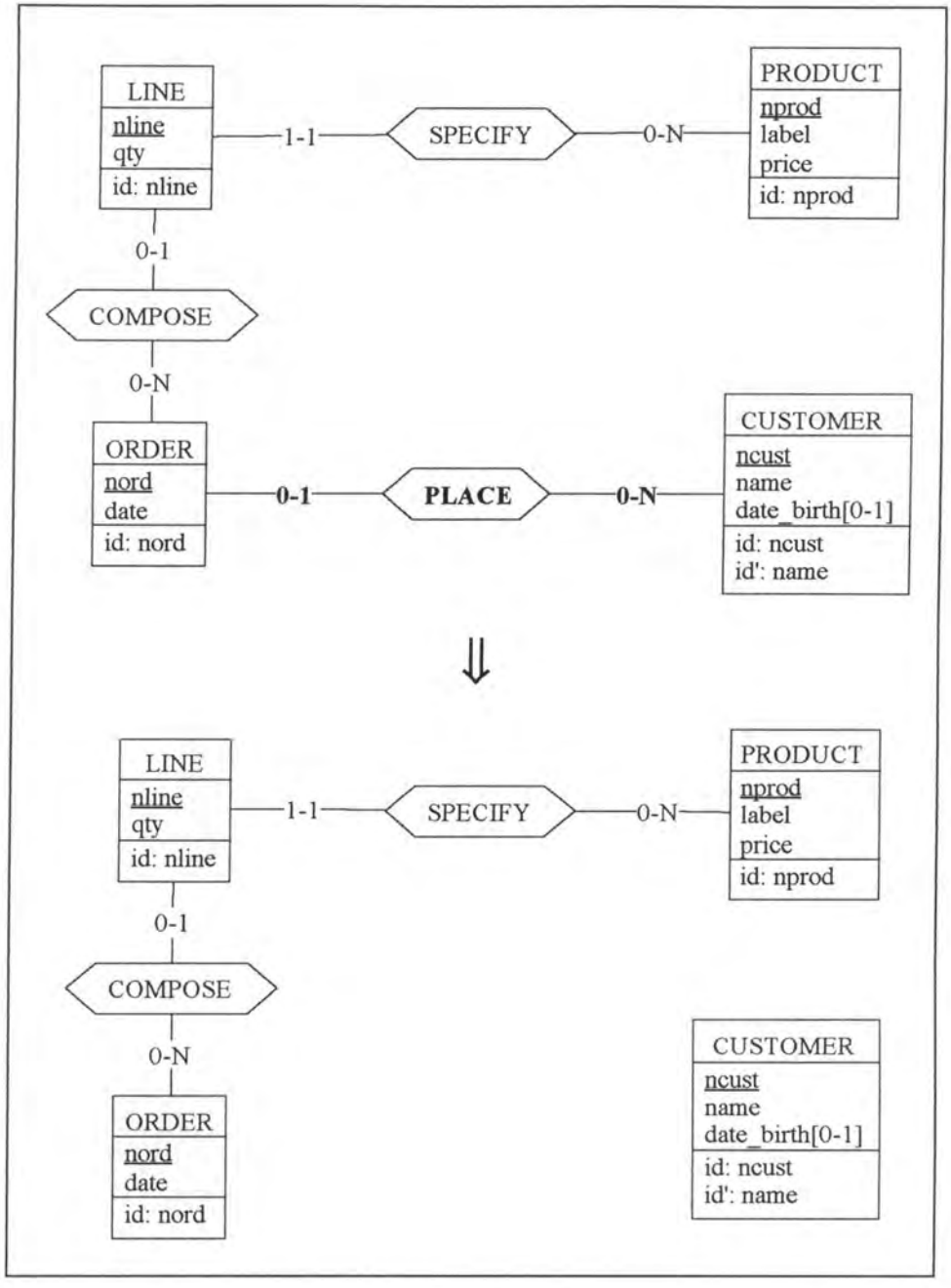


Figure A1 - 23 : Removing a 0-1/0-N relationship-type on the conceptual level

3.2.4.1. Logical Schema

We remove the column PLACE_ncust in ORDER with its foreign key feature.

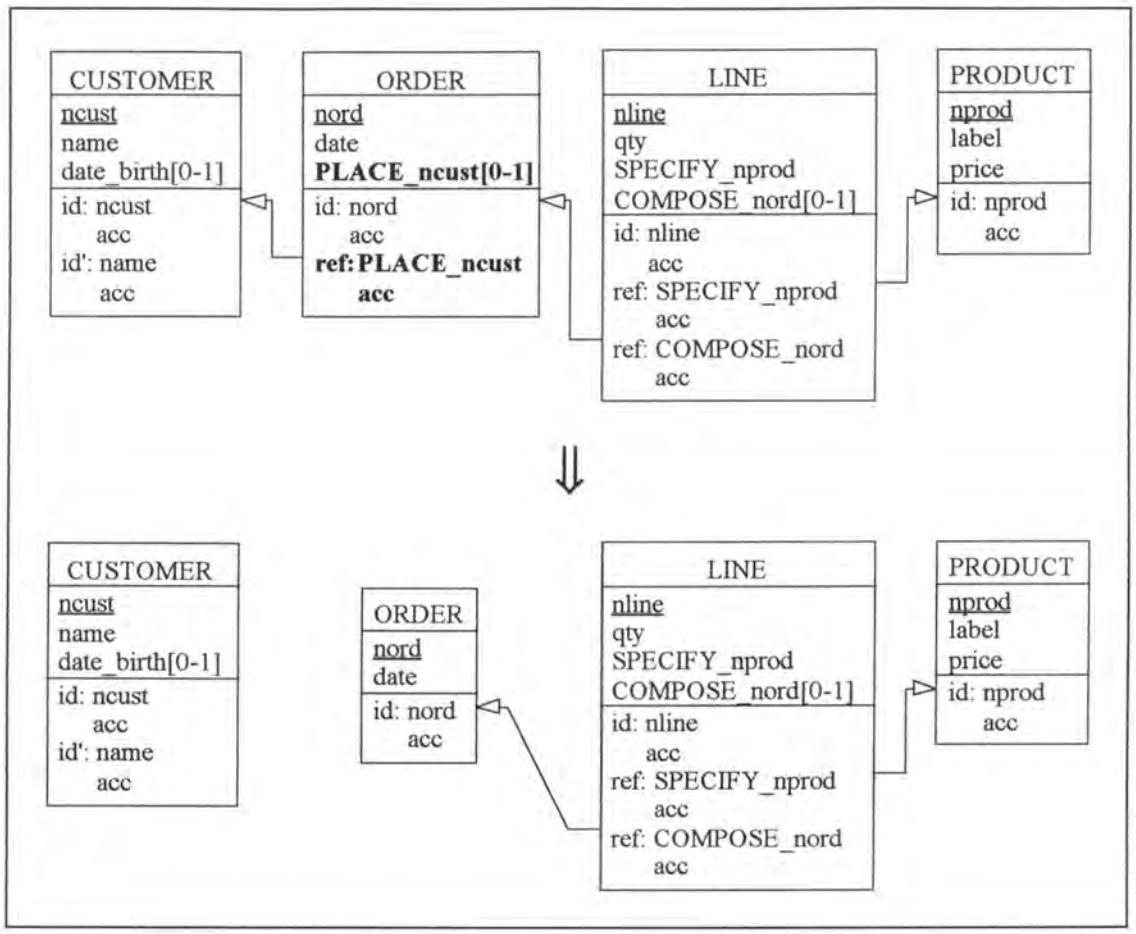


Figure A1 - 24 : Removing a 0-1/0-N relationship-type on the logical level

3.2.4.2. SQL Description & Data

```
alter table ORDER
  drop constraint CUS1, (* we remove the foreign key feature *)
  drop PLACE_ncust;
```

The link between an ORDER and its CUSTOMER is lost as shown in Figure A1-25.

ORDER	
nord	date
E386	02/01/1995
F285	12/03/1994
G274	15/07/1993
F842	31/12/1994
E345	05/01/1995
:	:

Figure A1 - 25 : Table ORDER when the link with table CUSTOMER is lost

3.2.4.3. Program Extracts

The second SELECT query of our case study (see page 4-7) must be dropped. Moreover, the JOIN query (see page 4-8) may be modified as follows:

```
select name
  from CUSTOMER
  where date_birth < 01/01/1977.
```

Concerning the application programs, a similar remark as for the case remove_1-1/0-1_rel-type (see page A1-23) can be formulated.

3.3. MODIFICATIONS WHICH PRESERVE THE SEMANTICS

3.3.1. Rename_1-1/0-1_rel-type

Let us suppose we want to rename the relationship-type LIVE into HOME.

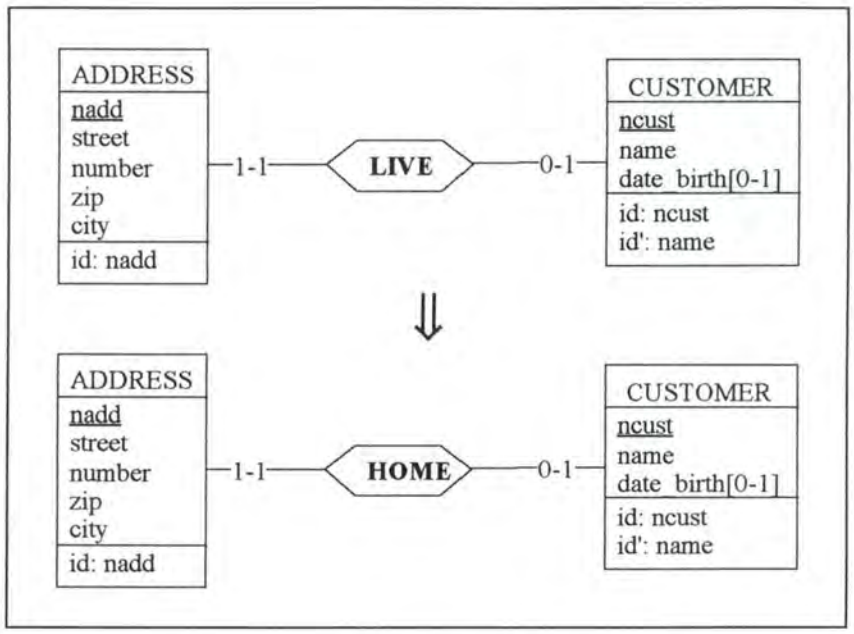


Figure A1 - 26 : Renaming a 1-1/0-1 relationship-type on the conceptual level

3.3.1.1. Logical Schema

On the logical level, we have to rename the foreign key column LIVE_ncust and its foreign and candidate key features.

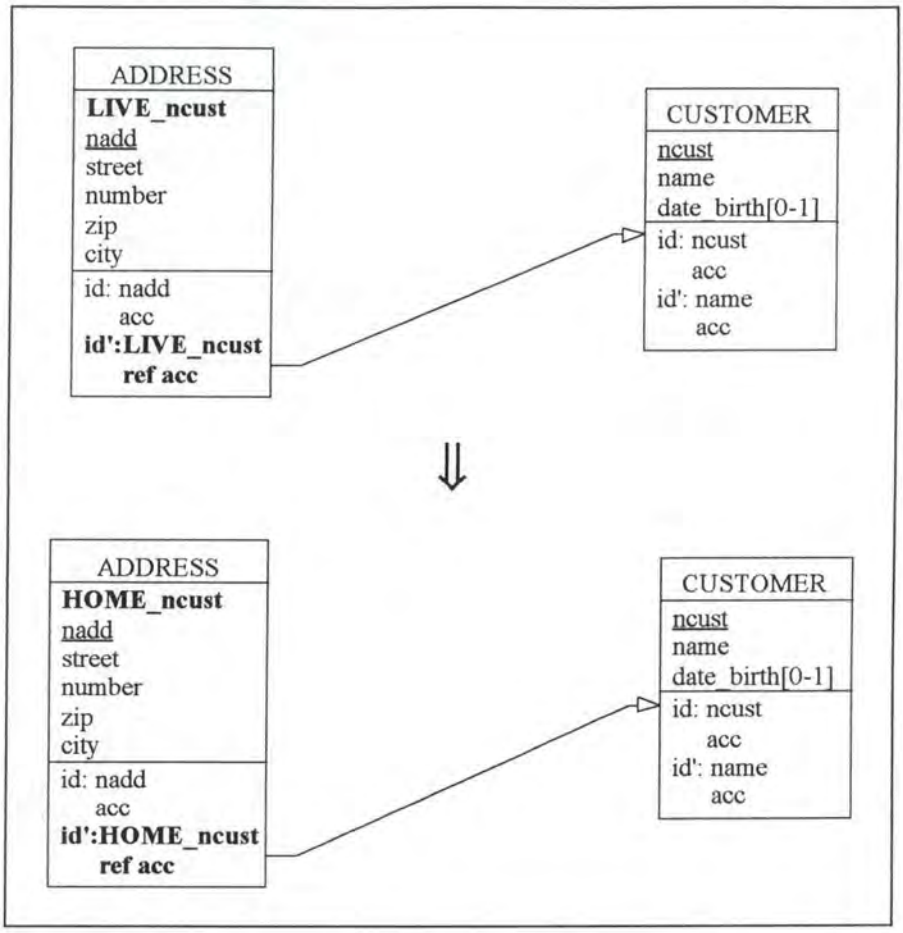


Figure A1 - 27 : Renaming a 1-1/0-1 relationship-type on the logical level

3.3.1.2. SQL Description & Data

```
alter table ADDRESS
  add HOME_ncust char(4) default '0000'  not null constraint A_HOME_ncust;
update ADDRESS
  set HOME_ncust = LIVE_ncust;
alter table ADDRESS
  drop constraint CUS1, (* we remove the old foreign key feature *)
  drop constraint idADD2,     (* we remove the old unique key feature *)
  drop constraint A_LIVE_ncust,     (* we remove the mandatory feature from
                                       column LIVE_ncust *)
  add constraint unique (HOME_ncust) constraint idADD2,
  add constraint foreign key (HOME_ncust)  references CUSTOMER
                                                  constraint CUS1,
  drop LIVE_ncust;
```

This operation does not involve loss of data as the values of column LIVE_ncust are copied into column HOME_ncust.

3.3.1.3. Program Extracts

We have to rename LIVE_ncust in all the select queries referencing it. In addition to the select queries, we have also to review the application programs in which they appear. For instance, we must rename certain variables and/or some fields or headings in the user interfaces.

3.3.2. Rename_0-1/0-1_rel-type

This case is similar to the case rename_1-1/0-1_rel-type (see page A1-32).

3.3.3. Rename_1-1/0-N_rel-type

Let us rename the 1-1/0-N relationship-type SPECIFY between LINE and PRODUCT into REFERENCE.

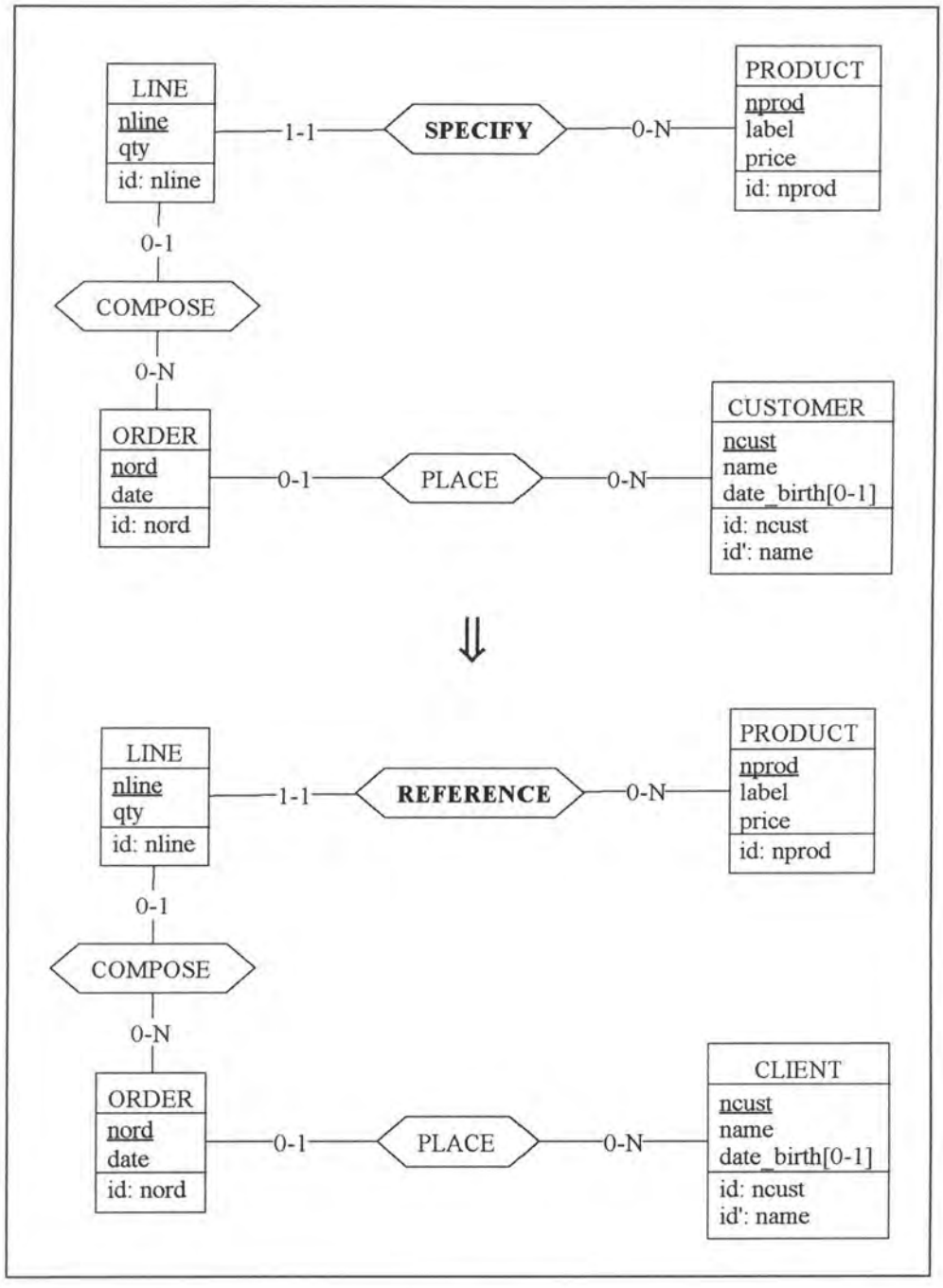


Figure A1 - 28 : Renaming a 1-1/0-N relationship-type on the conceptual level

3.3.3.1. Logical Schema

In the logical schema, the foreign key column SPECIFY_nprod and its feature must be renamed.

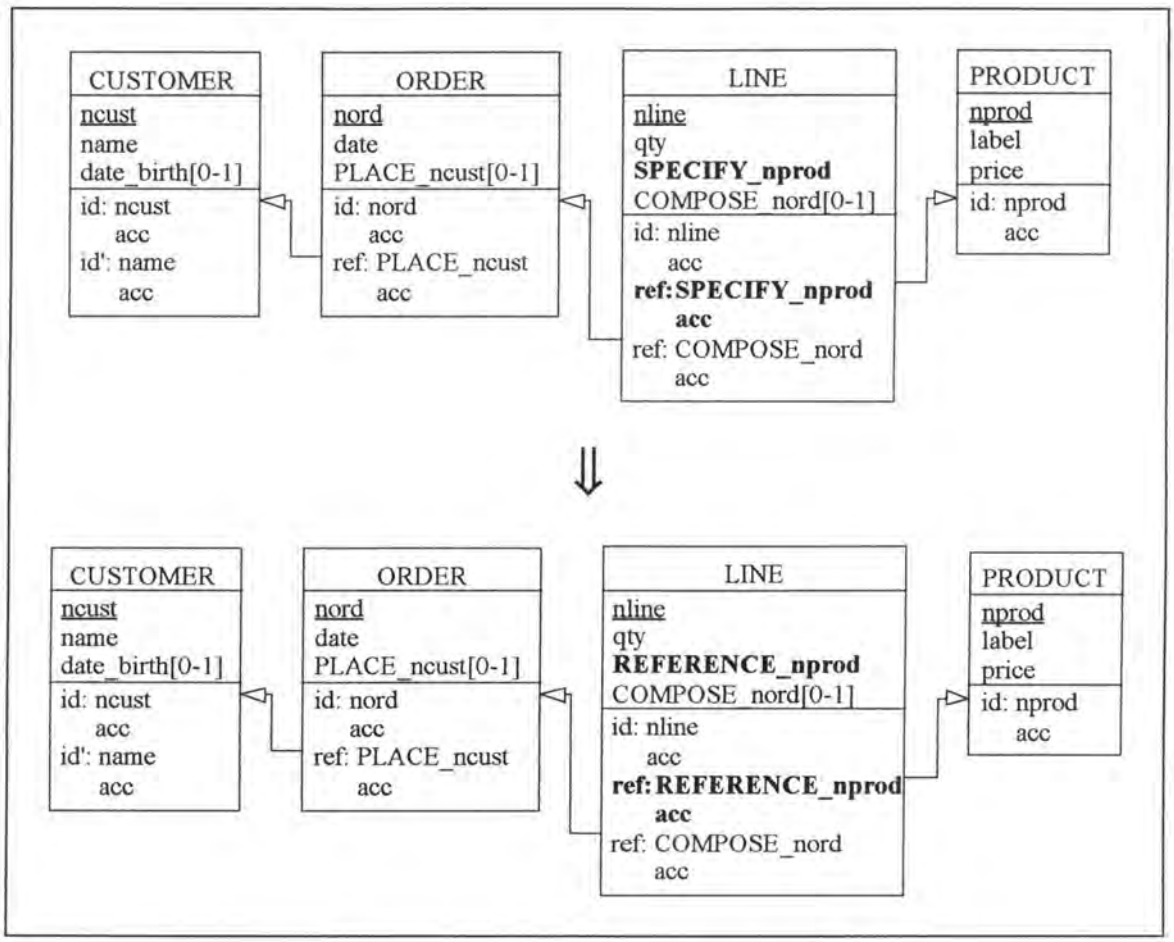


Figure A1 - 29 : Renaming a 1-1/0-N relationship-type on the logical level

3.3.3.2. SQL Description & Data

```
alter table LINE
  add REFERENCE_nprod  char(5)  default '00000'  not null
                                  constraint L_REFERENCE_nprod;
update LINE
  set REFERENCE_nprod = SPECIFY_nprod;
alter table LINE
  drop constraint PRO1, (* we remove the old foreign key feature *)
  drop constraint L_SPECIFY_nprod,  (* we remove the mandatory feature from
                                  column SPECIFY_nprod *)
  add constraint foreign key (REFERENCE_nprod) references PRODUCT
                                                  constraint PRO1,
  drop SPECIFY_nprod;
```

This operation does not involve loss of data as the values of column SPECIFY_nprod are copied into column REFERENCE_nprod.

3.3.3.3. Program Extracts

In all the select queries referencing SPECIFY_nprod we have to rename it. Concerning the application programs, a similar remark as for the case rename_1-1/0-1_rel-type can be formulated (see page A1-35).

3.3.4. Rename_0-1/0-N_rel-type

This case is similar to the case rename_1-1/0-N_rel_type (see page A1-35).

4. MODIFICATIONS OF THE ROLES

4.1. MODIFICATIONS WHICH AUGMENT THE SEMANTICS

4.1.1. Augment_max_card

Precondition:
Given the restrictions of the relationship-types in the Kernel (see page 3-2) the only augmentations of the maximum cardinality of a role that we accept so far are:

- 1-1/0-1 → 1-1/0-N
- 0-1/0-1 → 0-1/0-N

We consider an example for each of the two cases.

4.1.1.1. 1-1/0-1 → 1-1/0-N

Let us reconsider the example where a CUSTOMER LIVEs at an ADDRESS. We want to augment the maximum cardinality of the 0-1 role to N.

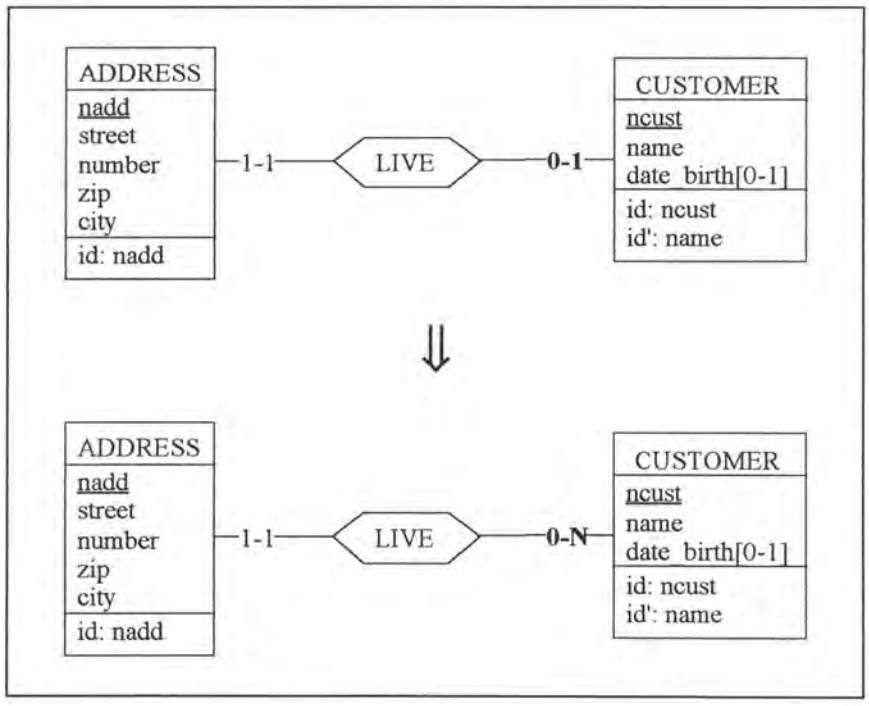


Figure A1 - 30 : Augmenting the maximum cardinality of a role to N in an 1-1/0-1 relationship-type on the conceptual level

4.1.1.1.1. Logical Schema

We have to remove the candidate key feature from the foreign key LIVE_ncust in relation ADDRESS.

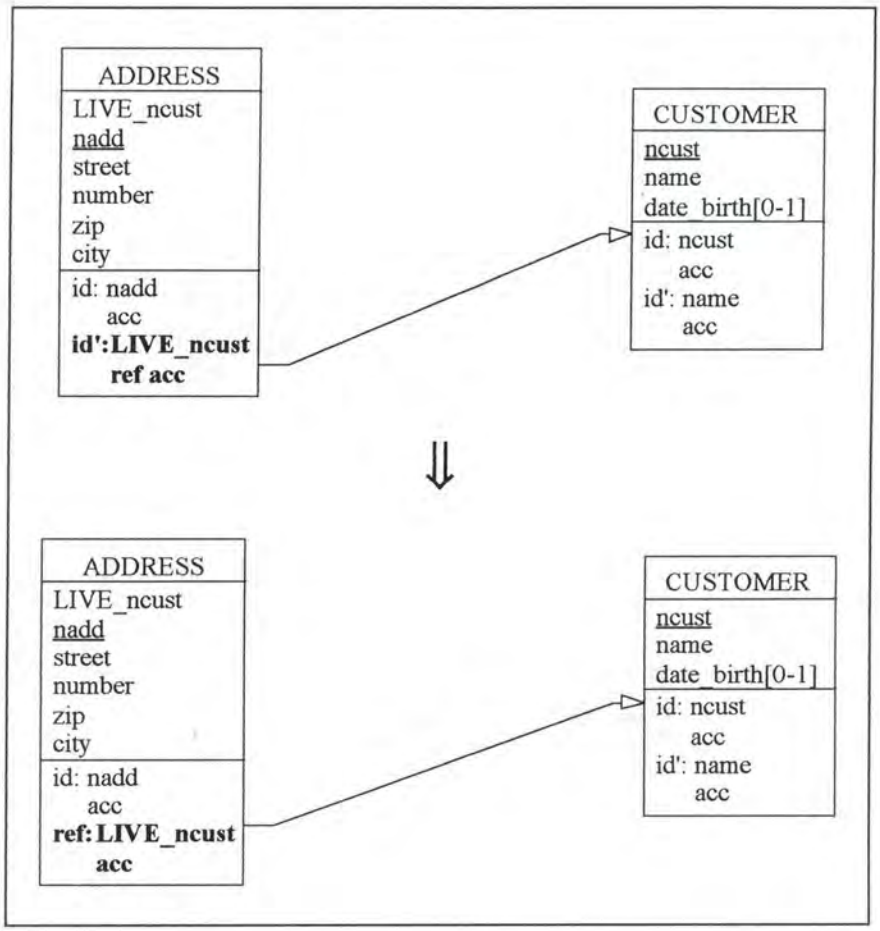


Figure A1 - 31 : Augmenting the maximum cardinality of a role to N in an 1-1/0-1 relationship-type on the logical level

4.1.1.1.2. SQL Description & Data

```
alter table ADDRESS
  drop constraint idADD2;
```

No modification is made on the data as we only remove a unique key constraint.

4.1.1.1.3. Program Extracts

Let us consider the following program extract:

```
var street: STRING[20];
    number: INTEGER;
    zip: INTEGER;
    city: STRING[20];
```

```
:
exec SQL
      select street, number, zip, city
        into :street, :number, :zip, :city
        from ADDRESS
        where LIVE_ncust = 'A101'
end exec;
if SQLCODE = 0    (* if such a row has been found *)
then begin
        write(street);
        write(number);
        write(zip);
        write(city);
     end;
:
```

We have to adapt this extract, as shown here below, in order to allow a CUSTOMER to have several ADDRESSes. Note that the simple treatment (if...then) has to be replaced by a loop treatment (while...do).

```
var street: STRING[20];
    number: INTEGER;
    zip: INTEGER;
    city: STRING[20];

   :
exec SQL
      declare c cursor for
        select street, number, zip, city
          from ADDRESS
          where LIVE_ncust = 'A101';
      open c;
      fetch c into :street, :number, :zip, :city;
end exec;
while SQLCODE = 0       (* the last item has not yet been treated *)
do begin
        write (street);
        write (number);
        write (zip);
        write (city);
        exec SQL
            fetch c into :street, :number, :zip, :city;
        end exec
     end;
exec SQL
      close c
end exec;
   :
```

In addition, let us reconsider the screen which displays the information about a CUSTOMER, including his/her working ADDRESS. As a CUSTOMER can now have several ADDRESSes, the user has to rearrange the screen so that it can display several ADDRESSes. Finally, the user has to replace certain variables by arrays.

4.1.1.2. 0-1/0-1 → 0-1/0-N

We want to transform the role of the relationship-type WORK played by CUSTOMER into 0-N.

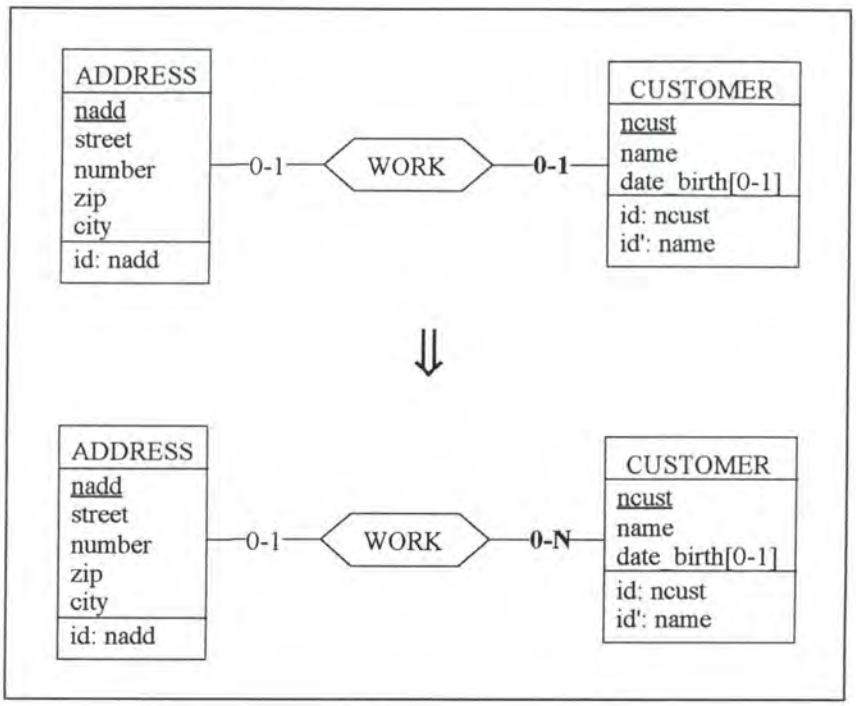


Figure A1 - 32 : Augmenting the maximum cardinality of a role to N in an 0-1/0-1 relationship-type on the conceptual level

There are two possible representations on the logical level:

- WORK is implemented by a foreign key in relation ADDRESS
- WORK is implemented by a foreign key in relation CUSTOMER

4.1.1.2.1. WORK is implemented by a foreign key in relation ADDRESS

4.1.1.2.1.1. Logical Schema

The initial logical schema is:

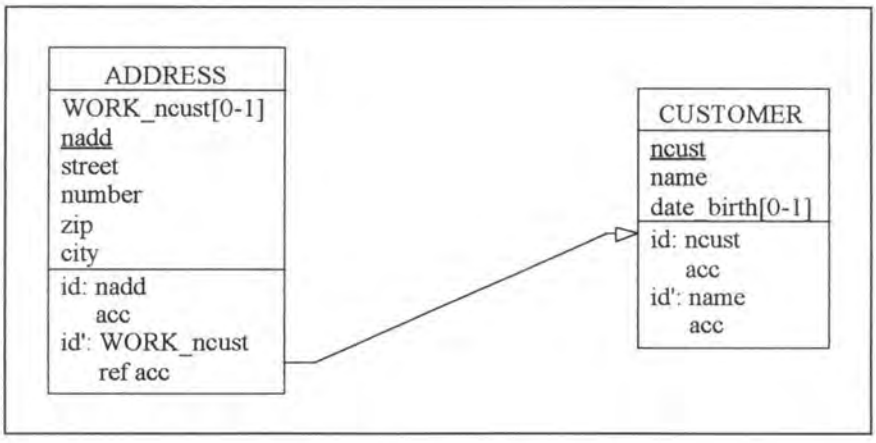


Figure A1 - 33 : The initial logical schema

This case is identical to the case 4.1.1.1. (see page A1-39).

4.1.1.2.2. WORK is implemented by a foreign key in relation CUSTOMER

4.1.1.2.2.1. Logical Schema

On the logical level, the transformation is:

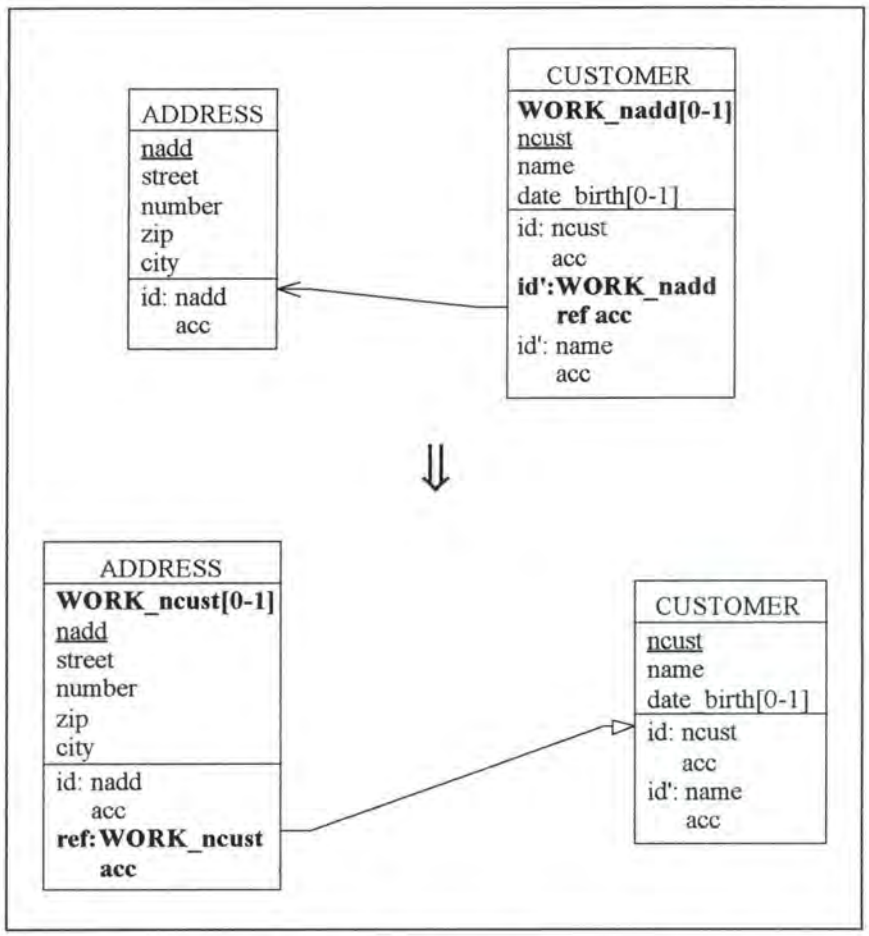


Figure A1 - 34 : Augmenting the maximum cardinality of a role to N in an 0-1/0-1 relationship-type on the logical level

4.1.1.2.2.2. SQL Description & Data

```
var cust: STRING[4];
    add: INTEGER;

exec SQL
      (* we create the new foreign key column *)
      alter table ADDRESS
        add WORK_ncust char(4);
      (* we copy the data representing relationship-type WORK from table
         CUSTOMER into table ADDRESS *)
      declare c cursor for
        select ncust, WORK_nadd
          from CUSTOMER
          where WORK_nadd is not null;
```

```
      open c;
      fetch c into :cust, :add;
end exec;
while SQLCODE = 0       (* the last item has not yet been treated *)
do begin
      exec SQL
           update ADDRESS
             set WORK_ncust = :cust
             where nadd = :add;
           fetch c into :cust, :add;
      end exec;
   end;
exec SQL
      (* we add and remove the necessary constraints *)
      alter table ADDRESS
        add constraint foreign key (WORK_ncust) references
                                                CUSTOMER constraint CUS1;
      alter table CUSTOMER
        drop constraint idCUS2,  (* we remove the old unique key feature *)
        drop constraint ADD1,   (* we remove the old foreign key feature *)
        drop WORK_nadd;
      close c;
end exec;
```

Note that no data is lost as the data representing relationship-type WORK is 'copied' from table CUSTOMER into table ADDRESS.

4.1.1.2.2.3. Program Extracts

Application programs referencing the foreign key representing relationship-type WORK must be reviewed. Two possible modifications are:

-

```
var add: STRING[12];

exec SQL
      select WORK_nadd
        into :add
        from CUSTOMER
        where name = 'Hasselhoff S.';
end exec;
if SQLCODE = 0          (* if such a row has been found *)
then write (add);
```

```
var add: STRING[12];

exec SQL
      declare c cursor for
        select nadd
          from ADDRESS
          where WORK_ncust in (select ncust
                               from CUSTOMER
                               where name = 'Hasselhoff S.')
      open c;
      fetch c into :add;
end exec;
while SQLCODE = 0       (* the last item has not yet been treated *)
do begin
      write (add);
      exec SQL
           fetch c into :add;
      end exec;
   end;
exec SQL
```

```
        close c
    end exec;
```

- ```
  select street, city
    from ADDRESS
    where nadd in (select WORK_nadd
                   from CUSTOMER
                   where name like '%Dupont%')
  ```

  ⇓

  ```
  select street, city
    from ADDRESS
    where WORK_ncust in (select ncust
                         from CUSTOMER
                         where name like '%Dupont%').
  ```

Concerning the application programs, similar remarks as for the case 4.1.1.1.3. (see page A1-40) can be formulted here.

4.1.2. Decrease_min_card

Precondition:
Given the restrictions of the relationship-types in the Kernel (see page 3-2) the only decreases of the minimum cardinality of a role that we accept so far are:

- 1-1/0-1 → 0-1/0-1
- 1-1/0-N → 0-1/0-N

4.1.2.1. 1-1/0-1 → 0-1/0-1

Let us reconsider the example where a CUSTOMER LIVEs at an ADDRESS. We want to decrease the minimum cardinality of the 1-1 role to 0.
  ```

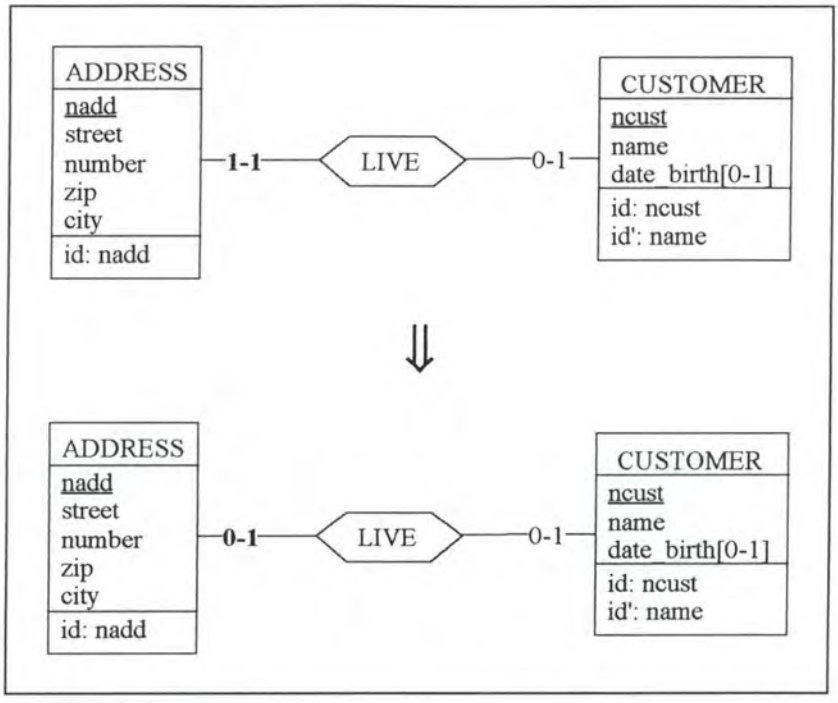


Figure A1 - 35 : Decreasing the minimum cardinality of a role to 0 in an 1-1/0-1 relationship-type on the conceptual level

4.1.2.1.1. Logical Schema

We have to make the foreign key LIVE_ncust in ADDRESS optional.

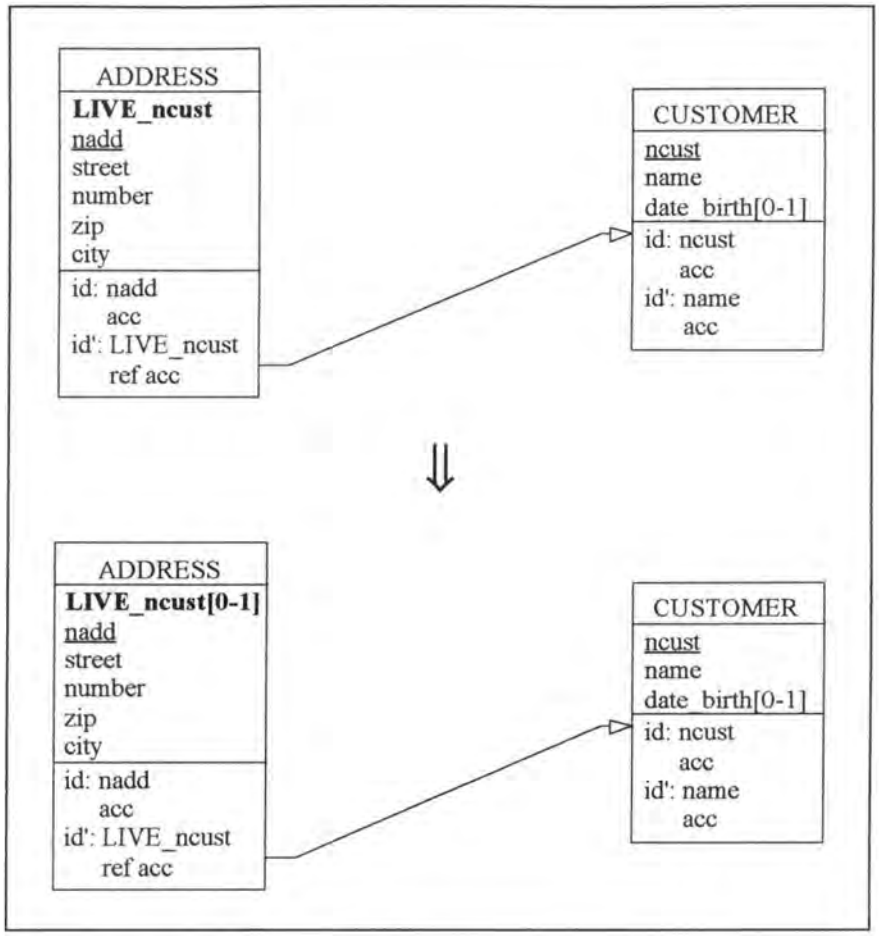


Figure A1 - 36 : Decreasing the minimum cardinality of a role to 0 in an 1-1/0-1 relationship-type on the logical level

4.1.2.1.2. SQL Description & Data

```
alter table ADDRESS
  drop constraint A_LIVE_ncust;
```

Note that no data is lost as the foreign key column is only made optional.

4.1.2.1.3. Program Extracts

Program extracts referencing the foreign key representing relationship-type LIVE must be reviewed. A possible modification would be:

```
var ncust: STRING[4];
    add: INTEGER;

  :
exec SQL
      select LIVE_ncust, nadd
        into :ncust, :add
        from ADDRESS
        where nadd = 110;
end exec;
```

```
if SQLCODE = 0          (* if such a row has been found *)
then write ('The customer living at address', add, 'is:', ncust)
else write ('The address', add, 'does not exist.');
  :
```

```
var ncust: STRING[4];
    null_indicator: INTEGER;
    add: INTEGER;

  :
exec SQL
     select LIVE_ncust, nadd
       into :ncust:null_indicator, :add
       from ADDRESS
       where nadd = 110;
end exec;
if SQLCODE = 0          (* if such a row has been found *)
then if null_indicator = 0    (* if the CUSTOMER is known *)
     then write ('The customer living at address', add, 'is:', ncust)
     else write (' No customer living at that address has been found.')
else write ('The address', add, 'does not exist.');
  :
```

As we can see in the previous program extracts, tests (if - then clauses) checking the null value of column LIVE_ncust must sometimes be introduced.

4.1.2.2. 1-1/0-N → 0-1/0-N

The case where we transform 1-1/0-N into 0-1/0-N is similar to the previous one (see page A1-45).

4.2. MODIFICATIONS WHICH DECREASE THE SEMANTICS

4.2.1. Decrease_max_card

Precondition:
Given the restrictions of the relationship-types in the Kernel (see page 3-2), the only decreases of the maximum cardinality of a role that we accept so far are:

- 1-1/0-N → 1-1/0-1
- 0-1/0-N → 0-1/0-1

We consider an example for each of the two cases:

4.2.1.1. 1-1/0-N → 1-1/0-1

Let us suppose that we want to decrease to 1 the maximum cardinality of the 0-N role of the relationship-type SPECIFY of our case study example.

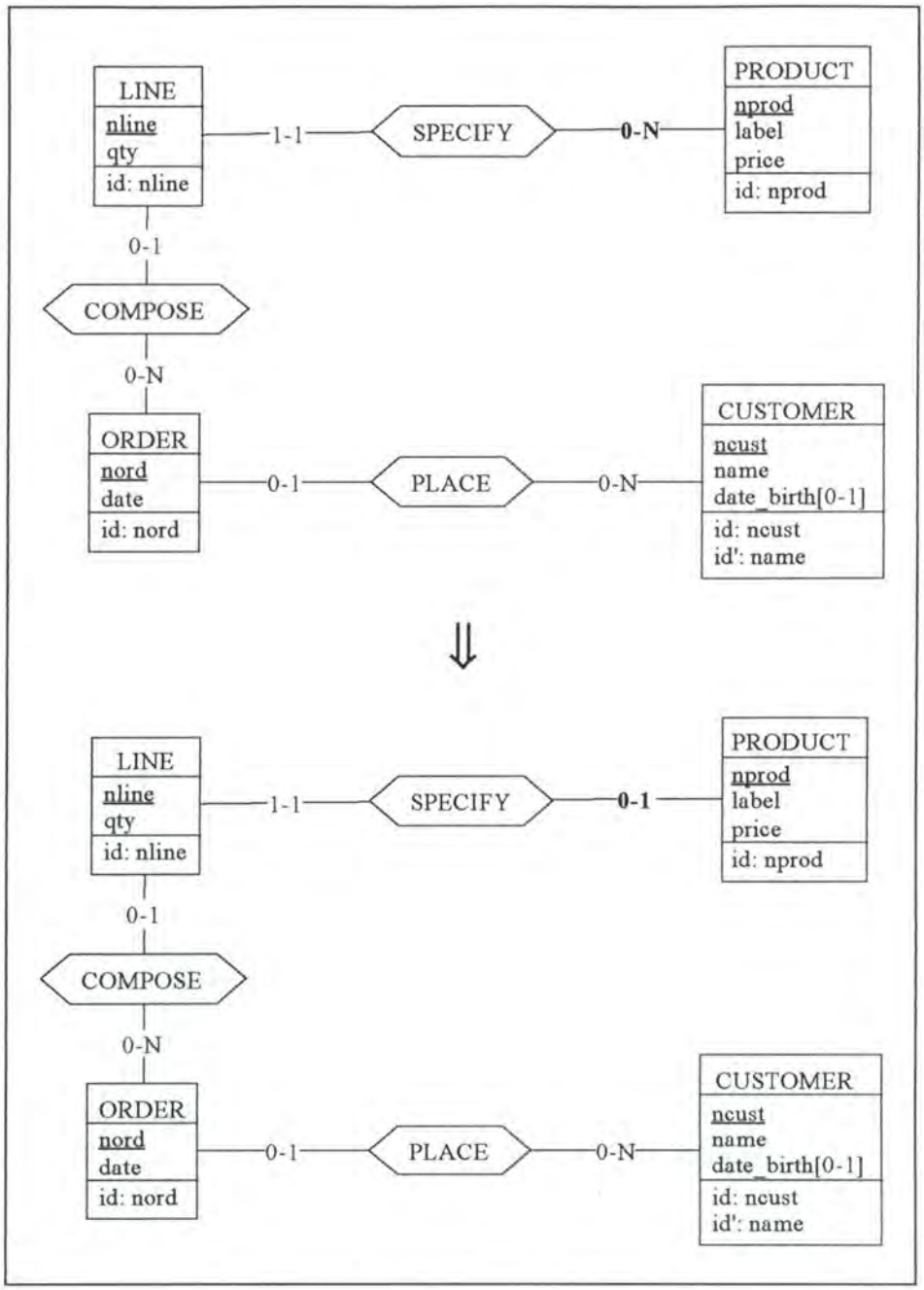


Figure A1 - 37 : Decreasing the maximum cardinality of a role in an 1-1/0-N relationship-type on the conceptual level

4.2.1.1.1. Logical Schema

On the logical level, we have to add the candidate key feature to column SPECIFY_nprod in relation LINE.

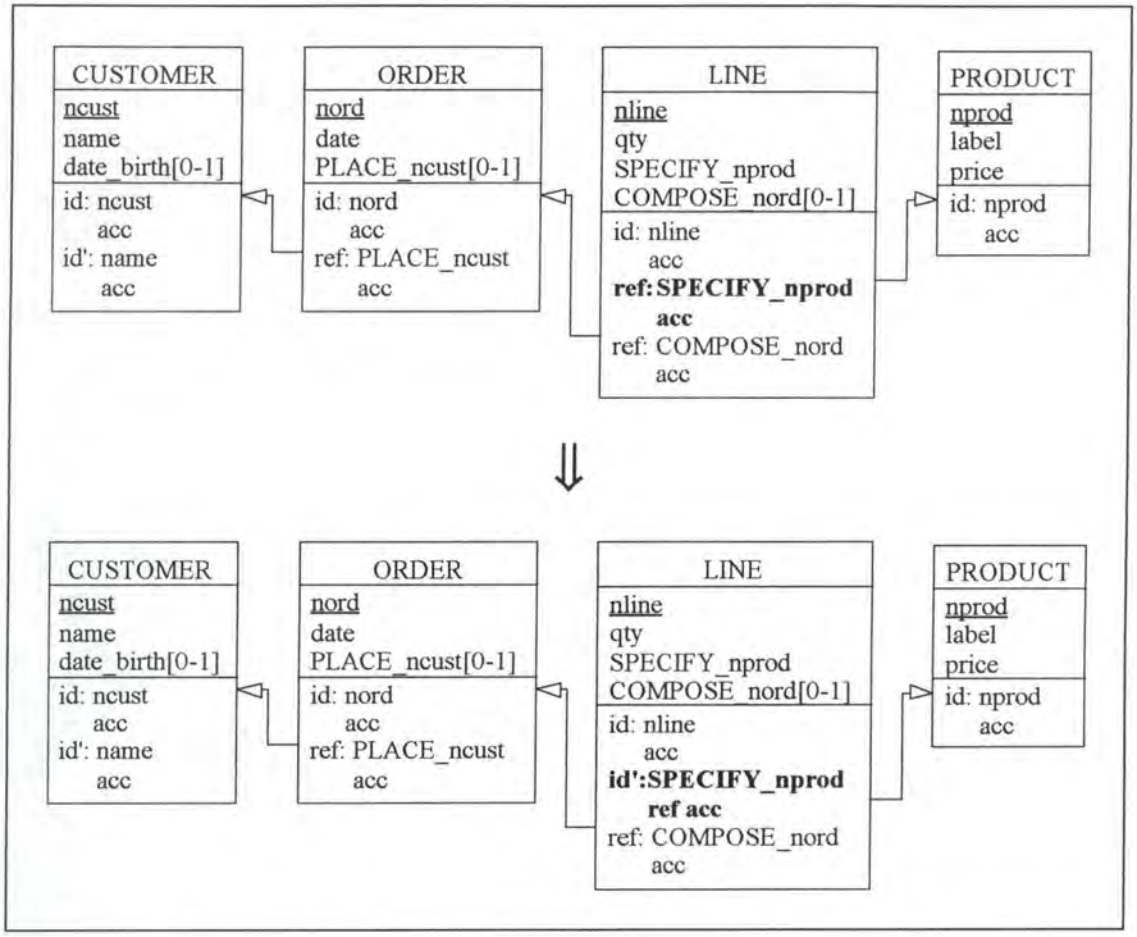


Figure A1 - 38 : Decreasing the maximum cardinality of a role in an 1-1/0-N relationship-type on the logical level

4.2.1.1.2. SQL Description & Data

```
var prod1: STRING[5];
    prod2: STRING[5];
    line: STRING[6];
    count: INTEGER;

exec SQL
     (* we have to delete all the rows except one of table LINE among
        those having the same value for SPECIFY_nprod *)
     declare c1 cursor for
       select SPECIFY_nprod, count(*)
         from LINE
         group by SPECIFY_nprod
         having count(*) > 1
         order by SPECIFY_nprod ASC;
     declare c2 cursor for
       select SPECIFY_nprod, nline
         from LINE
         group by SPECIFY_nprod, nline
         order by SPECIFY_nprod ASC, nline ASC;
     open c1;
     open c2;
     fetch c2 into :prod2, :line;
     fetch c1 into :prod1, :count;
```

```
end exec;
while SQLCODE = 0       (* the last item has not yet been treated *)
do begin
      while prod1 <> prod2
      do exec SQL
            fetch c2 into :prod2, :line
         end exec;
      exec SQL
            fetch c2 into :prod2, :line
      end exec;
      while (prod1 = prod2) and (SQLCODE = 0)
      do begin
            exec SQL
                  delete from LINE
                    where nline = :line;
                  fetch c2 into :prod2, :line;
            end exec;
         end;
      exec SQL
            fetch c1 into :prod1, :count
      end exec;
   end;
exec SQL
      close c1;
      close c2;
      (* we add the unique key feature to column SPECIFY_nprod *)
      alter table LINE
        add constraint unique (SPECIFY_nprod) constraint idLIN2;
end exec;
```

Each value in column SPECIFY_nprod in relation LINE must be unique. We can for example keep only one of the two following rows:

nline	(COMPOSE_nord)	qty	SPECIFY_nprod
ER5678	null	4587	EG880
DS5432	E583	5698	EG880

The way in which the modification is implemented would result in the loss of the first row. The modification for the whole table LINE is depicted in Figure A1-39.

LINE			
nline	(COMPOSE_nord)	qty	SPECIFY_nprod
AB1234	E386	1000	AA110
GH2345	null	1518	CA510
RT3456	**F285**	**345**	**AA110**
ZU4567	G274	2536	BE072
ER5678	**null**	**4587**	**EG880**
NM6789	G274	5558	WN592
OP7890	G274	5458	RK560
JK0987	F842	5473	SW226
TZ9876	E386	623	LS906
KJ8765	null	4587	SG953
WQ7654	F902	6325	BY907
XY6543	**null**	**9658**	**BY907**
DS5432	E583	5698	EG880
BV1357	F842	7458	AB099
IO2468	**G809**	**4125**	**AB099**

LINE.COMPOSE_nord in ORDER.nord
LINE.SPECIFY_nprod in PRODUCT.nprod

⇓

LINE			
nline	(COMPOSE_nord)	qty	SPECIFY_nprod
AB1234	E386	1000	AA110
GH2345	null	1518	CA510
ZU4567	G274	2536	BE072
NM6789	G274	5558	WN592
OP7890	G274	5458	RK560
JK0987	F842	5473	SW226
TZ9876	E386	623	LS906
KJ8765	null	4587	SG953
WQ7654	F902	6325	BY907
DS5432	E583	5698	EG880
BV1357	F842	7458	AB099

LINE.COMPOSE_nord in ORDER.nord
LINE.SPECIFY_nprod in PRODUCT.nprod

Figure A1 - 39 : The modification decrease_max_card applied on column SPECIFY_nprod of table LINE

4.2.1.1.3. Program Extracts

> ***Note:***
> The modifications suggested here below are not absolutely necessary. They may be seen as optimisations.

The optimisation would consist in replacing certain loops (for example while-loops) by simple if-then tests.

```
var qty: INTEGER;

 :
exec SQL
      declare c cursor for
        select qty
          from LINE
          where SPECIFY_nprod = 'AA110';
      open c;
      fetch c into :qty;
end exec;
while SQLCODE = 0
do begin
      write(qty);
      exec SQL
            fetch c into :qty
      end exec;
   end;
exec SQL
      close c
end exec;
 :
```

```
var qty: INTEGER;

 :
exec SQL
      select qty
        into :qty
        from LINE
        where SPECIFY_nprod = 'AA110';
end exec;
if SQLCODE = 0
then write(qty);
 :
```

Note that the user interfaces must also be adapted.

4.2.1.2. 0-1/0-N → 0-1/0-1

Let us suppose that we want to decrease to 1 the maximum cardinality of the 0-N role of the relationship-type PLACE of our case study example.

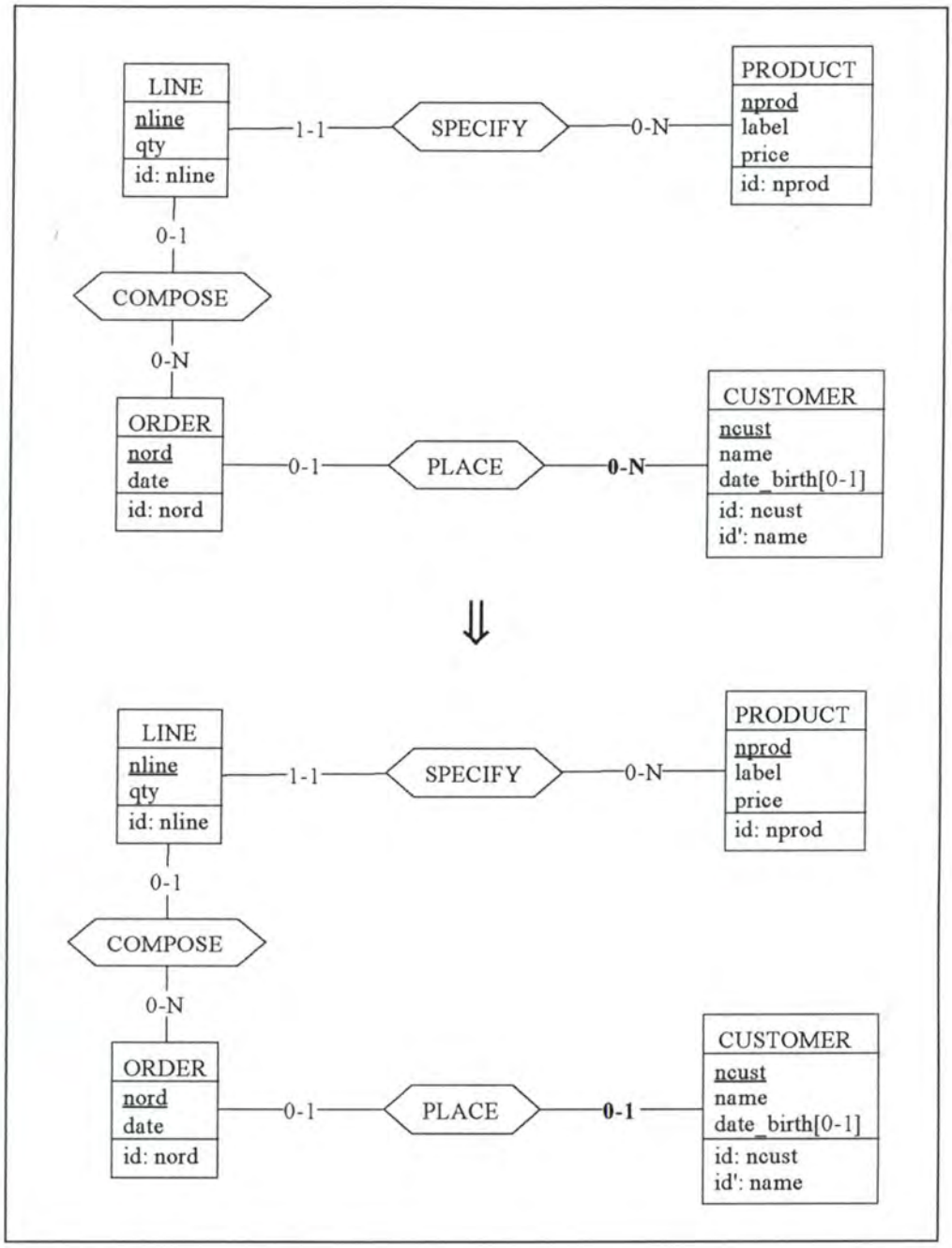


Figure A1 - 40 : Decreasing the maximum cardinality of a role in an 0-1/0-N relationship-type on the conceptual level

4.2.1.2.1. Logical Schema

On the logical level, we have to add the candidate key feature to column PLACE_ncust in relation ORDER.

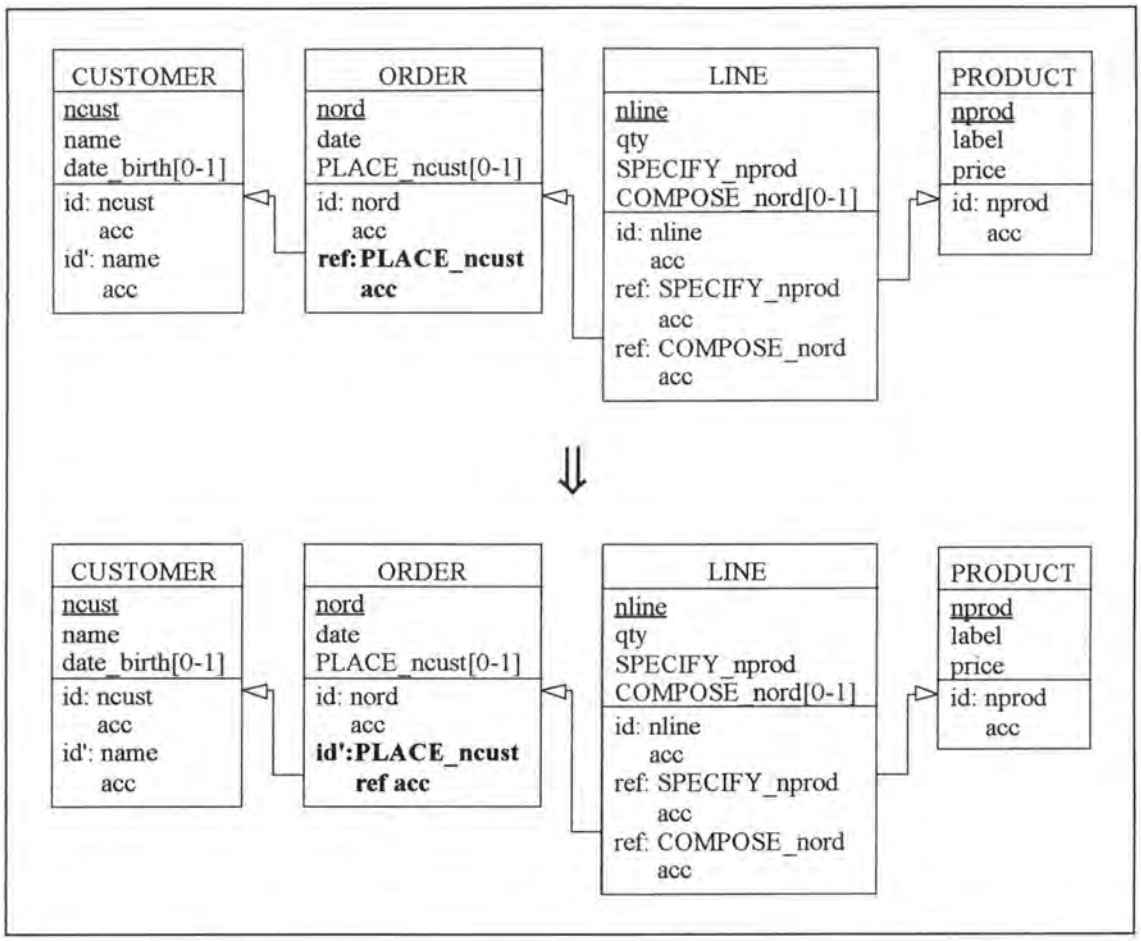


Figure A1 - 41 : Decreasing the maximum cardinality of a role in an 0-1/0-N relationship-type on the logical level

4.2.1.2.2. SQL Description & Data

We can proceed by two possible ways:

4.2.1.2.2.1. Deleting duplicate values of PLACE_ncust

We will proceed almost the same way as in the case 4.2.1.1.2. (see page A1-50). We will put in bold the additional operations.

```
var cust1: STRING[4];
    cust2: STRING[4];
    nord: STRING[4];
    count: INTEGER;

exec SQL
      (* we have to delete all the rows except one of table ORDER among
         those having the same value for PLACE_ncust *)
      declare c1 cursor for
        select PLACE_ncust, count(*)
          from ORDER
          group by PLACE_ncust
          having count(*) > 1
          order by PLACE_ncust ASC;
```

```
      declare c2 cursor for
        select PLACE_ncust, nord
          from ORDER
          group by PLACE_ncust, nord
          order by PLACE_ncust ASC, nord ASC;
      open c1;
      open c2;
      fetch c2 into :cust2, :nord;
      fetch c1 into :cust1, :count;
end exec;
while SQLCODE = 0       (* the last item has not yet been treated *)
do begin
      while cust1 <> cust2
      do exec SQL
            fetch c2 into :cust2, :nord
         end exec;
      exec SQL
            fetch c2 into :cust2, :nord
      end exec;
      while (cust1 = cust2) and (SQLCODE = 0)
      do begin
            exec SQL
                  (*we have to remove first the rows of table LINE
                    referencing the rows of table ORDER that will be
                    deleted*)
                  delete from LINE
                    where COMPOSE_nord = :nord;
                  delete from ORDER
                  where nord = :nord;
                  fetch c2 into :cust2, :nord;
            end exec;
         end;
      exec SQL
            fetch c1 into :cust1, :count
      end exec;
   end;
exec SQL
      close c1;
      close c2;
      (* we add the unique key feature to column PLACE_ncust *)
      alter table ORDER
        add constraint unique (PLACE_ncust) constraint idORD2;
end exec;
```

In table ORDER, PLACE_ncust will become a unique key. The resulting table is shown in Figure A1-42.

ORDER		
nord	(PLACE_ncust)	date
E386	A958	02/01/1995
G274	null	15/07/1993
F842	C395	31/12/1994
E345	B234	05/01/1995
F902	D365	16/09/1994
E583	B472	12/01/1995
F676	A101	26/02/1993
G809	null	23/05/1994

ORDER.PLACE_ncust in CUSTOMER.ncust

Figure A1 - 42 : The table ORDER after having removed duplicate values for PLACE_ncust

This modification results in the loss of the two following rows of table ORDER:

nord	(PLACE_ncust)	date
F285	B472	12/03/1994
G222	A958	23/05/1994

This loss of data has an immediate impact on table LINE. We loose the following row in table LINE:

nline	(COMPOSE_nord)	qty	SPECIFY_nprod
RT3456	*F285*	345	AA110

4.2.1.2.2.2. Setting duplicate values of column PLACE_ncust to null

We proceed almost the same way as in case 4.2.1.1.2. (see page A1-50). We will put in bold the operations that will change.

```
var cust1: STRING[4];
    cust2: STRING[4];
    nord: STRING[4];
    count: INTEGER;

exec SQL
      (* we have to remove all the rows except one of table ORDER among
         those having the same value for PLACE_ncust *)
      declare c1 cursor for
        select PLACE_ncust, count(*)
          from ORDER
          group by PLACE_ncust
          having count(*) > 1
          order by PLACE_ncust ASC;
      declare c2 cursor for
        select PLACE_ncust, nord
          from ORDER
          group by PLACE_ncust, nord
          order by PLACE_ncust ASC, nord ASC;
```

```
      open c1;
      open c2;
      fetch c2 into :cust2, :nord;
      fetch c1 into :cust1, :count;
end exec;
while SQLCODE = 0       (* the last item has not yet been treated *)
do begin
      while cust1 <> cust2
      do exec SQL
            fetch c2 into :cust2, :nord
         end exec;
      exec SQL
            fetch c2 into :cust2, :nord
      end exec;
      while (cust1 = cust2) and (SQLCODE = 0)
      do begin
            exec SQL
                  update ORDER
                    set PLACE_ncust = null
                    where nord = :nord;
                  fetch c2 into :cust2, :nord;
            end exec;
         end;
      exec SQL
            fetch c1 into :cust1, :count
      end exec;
   end;
exec SQL
      close c1;
      close c2;
      (* we add the unique key feature to column PLACE_ncust *)
      alter table ORDER
        add constraint unique(PLACE_ncust) constraint idORD2;
end exec;
```

Each value in column PLACE_ncust in relation ORDER must be unique (except for the null value). We can this time keep the two following rows:

nord	(PLACE_ncust)	date
E386	A958	02/01/1995
G222	A958	23/05/1994

by setting the PLACE_ncust value of the second row to null:

nord	(PLACE_ncust)	date
E386	A958	02/01/1995
G222	null	23/05/1994

We put in bold the modifications on table ORDER.

ORDER		
nord	(PLACE_ncust)	date
E386	A958	02/01/1995
F285	**null**	12/03/1994
G274	null	15/07/1993
F842	C395	31/12/1994
E345	B234	05/01/1995
G222	**null**	23/05/1994
F902	D365	16/09/1994
E583	B472	12/01/1995
F676	A101	26/02/1993
G809	null	23/05/1994

ORDER.PLACE_ncust in CUSTOMER.ncust

Figure A1 - 43 : Table ORDER after having set duplicate values of PLACE_ncust to null

4.2.1.2.3. Program Extracts

The same suggestions as in the case 4.2.1.1.3 (see page A1-53) can be made here.

4.2.2. Augment_min_card

Precondition:
Given the restrictions of the relationship-types in the Kernel (see page 3-2), the only augmentations of the minimum cardinality of a role that we accept so far are:

- 0-1/0-1 → 1-1/0-1
- 0-1/0-N → 1-1/0-N

We consider an example for each of the two cases.

4.2.2.1. 0-1/0-1 → 1-1/0-1

Let us reconsider the example where a CUSTOMER WORKs at an ADDRESS. We want to augment the minimum cardinality of the role played by ADDRESS to 1.

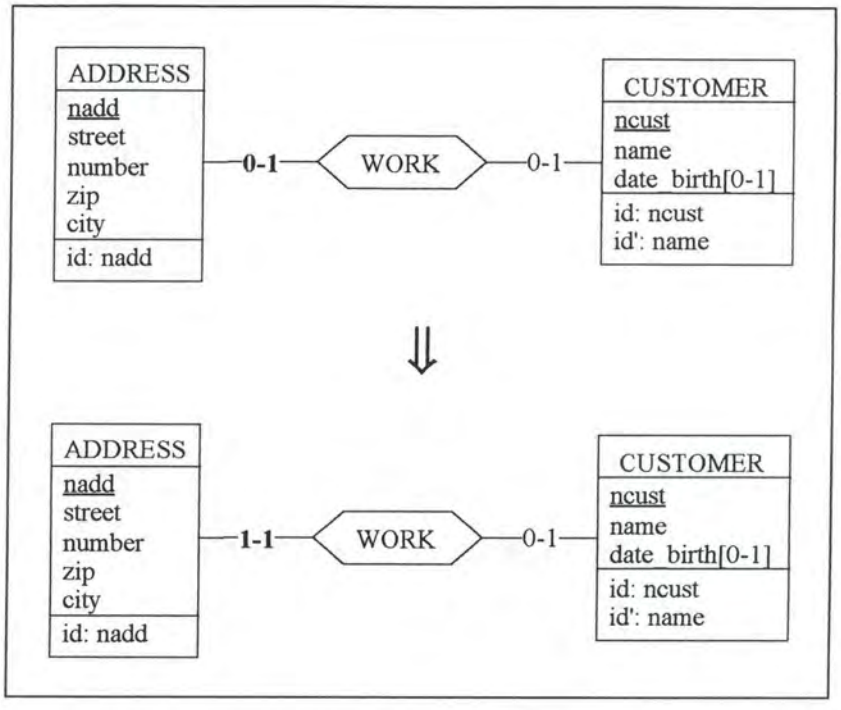


Figure A1 - 44 : Augmenting the minimum cardinality of a role to 1 in an 0-1/0-1 relationship-type on the conceptual level

Two different cases must be considered:

- WORK is implemented by a foreign key in ADDRESS
- WORK is implemented by a foreign key in CUSTOMER

4.2.2.1.1. WORK is implemented by a foreign key in ADDRESS

4.2.2.1.1.1. Logical Schema

We have to make the foreign key WORK_ncust in ADDRESS mandatory.

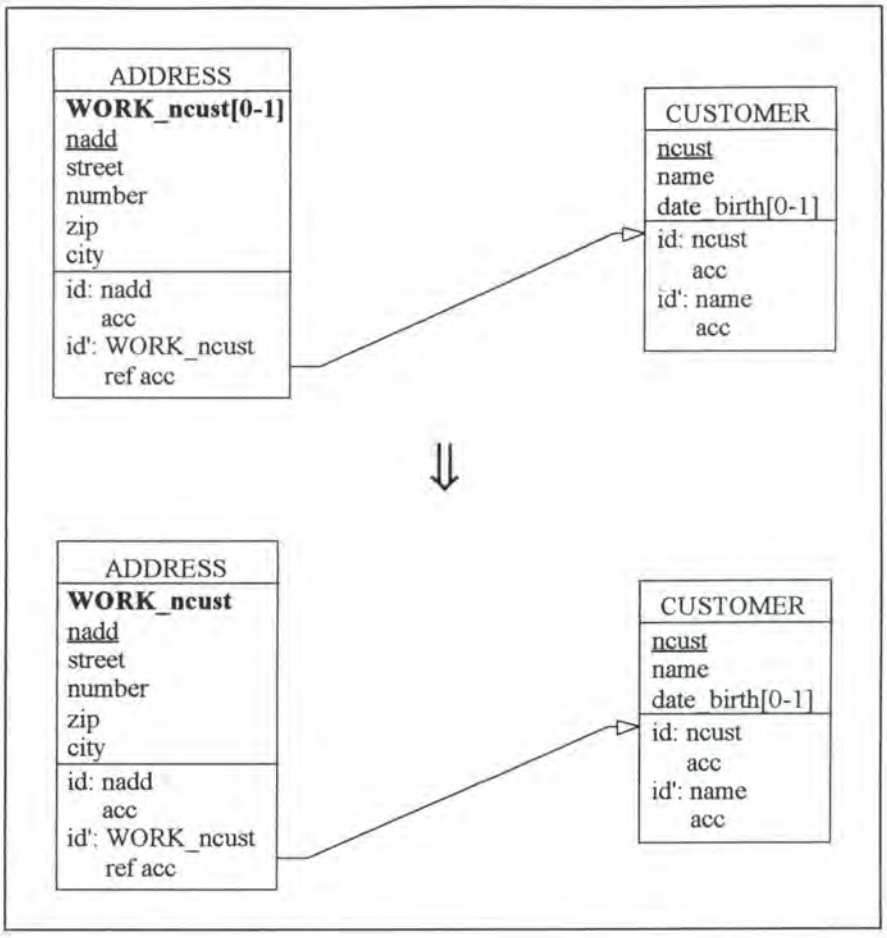


Figure A1 - 45 : Augmenting the minimum cardinality of a role to 1 in an 0-1/0-1 relationship-type on the logical level

4.2.2.1.1.2. SQL Description & Data

```
delete from ADDRESS
  where WORK_ncust is null;
alter table ADDRESS
  alter WORK_ncust  not null  constraint A_WORK_ncust;
```

This way of implementing the modification involves loss of data, as we drop the rows having a null value for column WORK_ncust.

4.2.2.1.1.3. Program Extracts

As we already said it is often not sufficient to delete or modify the select queries referencing the null value of WORK_ncust. The application programs in which they appear must also be reviewed. A possible modification would be:

```
var ncust: STRING[4];
    null_indicator: INTEGER;
    add: INTEGER;

  :
exec SQL
      select WORK_ncust, nadd
```

```
        into :ncust:null_indicator, :add
        from ADDRESS
        where nadd = 110;
end exec;
if SQLCODE = 0          (* if such a row has been found *)
then if null_indicator = 0    (*if the CUSTOMER is known*)
     then write ('The customer working at address', add, 'is:', ncust)
     else write (' No customer working at that address has been found.')
else write ('The address', add, 'does not exist.');
  :
```

```
var ncust: STRING[4];
    add: INTEGER;

  :
exec SQL
      select WORK_ncust, nadd
        into :ncust, :add
        from ADDRESS
        where nadd = 110;
end exec;
if SQLCODE = 0          (* if such a row has been found *)
then write ('The customer working at address', add, 'is:', ncust)
else write ('The address', add, 'does not exist.');
  :
```

4.2.2.1.2. WORK is implemented by a foreign key in CUSTOMER

4.2.2.1.2.1. Logical Schema

On the logical level the transformation is:

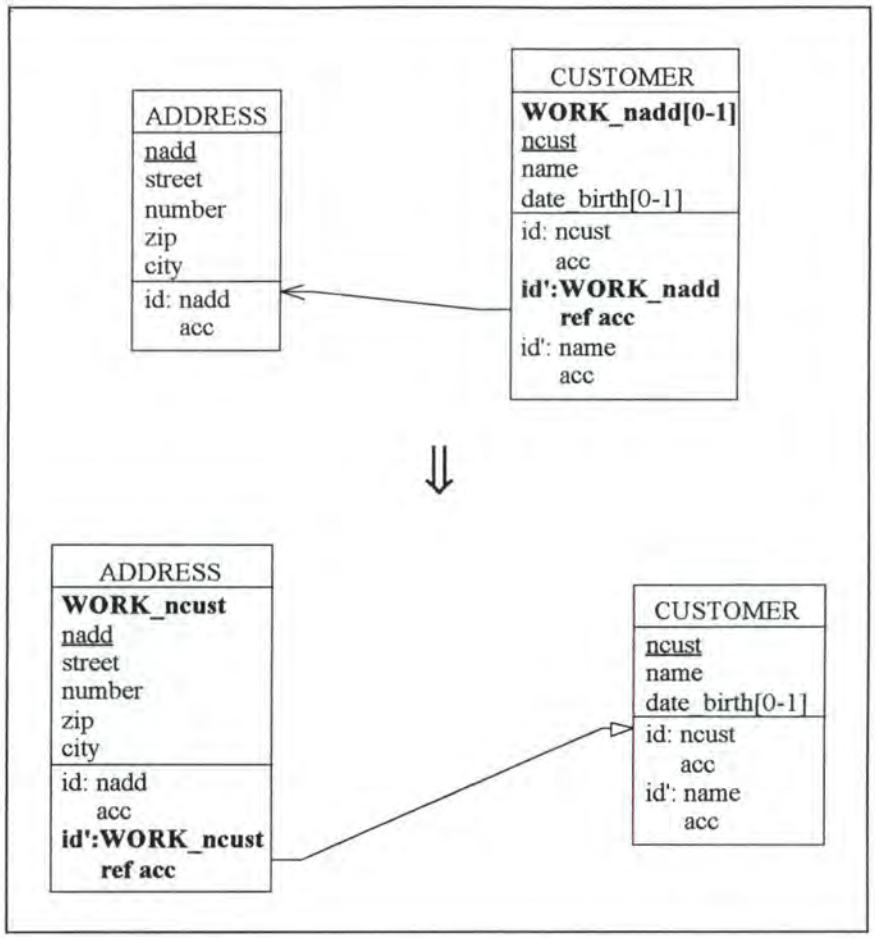


Figure A1 - 46 : Augmenting the minimum cardinality of a role to 1 in an 0-1/0-1 relationship-type on the logical level

4.2.2.1.2.2. SQL Description & Data

```
var  cust: STRING[4];
     add: INTEGER;

exec SQL
      (* we create the new foreign key column *)
      alter table ADDRESS
        add WORK_ncust  char(4)  default '0000'  not null
                                                 constraint A_WORK_ncust;
      (* we copy the data representing relationship-type WORK from table
         CUSTOMER into table ADDRESS *)
      declare c cursor for
        select ncust, WORK_nadd
          from CUSTOMER
          where WORK_nadd is not null;
      open c;
      fetch c into :cust, :add;
end exec;
while SQLCODE = 0       (* the last item has not yet been treated *)
do begin
      exec SQL
           update ADDRESS
             set WORK_ncust = :cust
             where nadd = :add;
           fetch c into :cust, :add;
```

```
      end exec;
   end;
exec SQL
      close c;
      (*we drop the rows from ADDRESS which are not linked to a CUSTOMER*)
      delete from ADDRESS
        where WORK_ncust = '0000';
      (* we add and remove the necessary constraints *)
      alter table ADDRESS
        add constraint unique (WORK_ncust) constraint idADD2,
        add constraint foreign key (WORK_ncust) references CUSTOMER
                                                    constraint CUS1;
      alter table CUSTOMER
        drop constraint idCUS3,  (* we remove the old unique key feature *)
        drop constraint ADD1, (* we remove the old foreign key feature*)
        drop WORK_nadd;
end exec
```

This way of implementing the modification involves loss of data, as we drop the rows from ADDRESS which are not linked to a CUSTOMER.

4.2.2.1.2.3. Program Extracts

Application programs referencing the foreign key representing relationship-type WORK must be reviewed. In some cases, we have to drop or change extracts in which select queries reference the null value of WORK_nadd, in other cases we have to change the extracts or queries referencing WORK_nadd in CUSTOMER. For example:

```
select name
  from CUSTOMER
  where WORK_nadd = 102;
```

```
select name
  from CUSTOMER
  where ncust in ( select WORK_ncust
                   from ADDRESS
                   where nadd = 102 )
```

4.2.2.2. 0-1/0-N → 1-1/0-N

Let us augment the minimum cardinality of the 0-1 role of relationship-type PLACE in our case study.

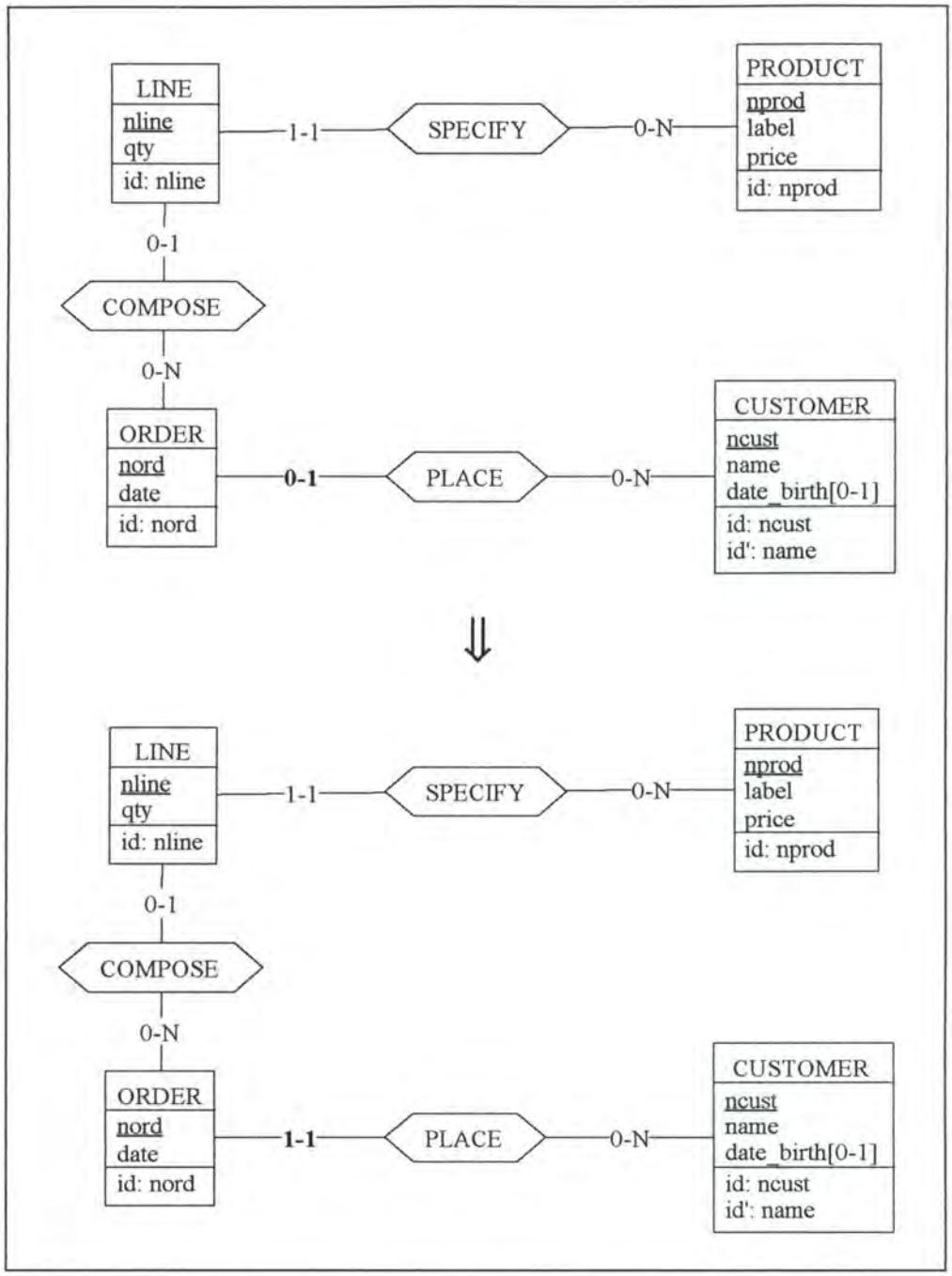


Figure A1 - 47 : Augmenting the minimum cardinality of a role to 1 in an 0-1/0-N relationship-type on the conceptual level

4.2.2.2.1. Logical Schema

We have to make the foreign key PLACE_ncust in ORDER mandatory.

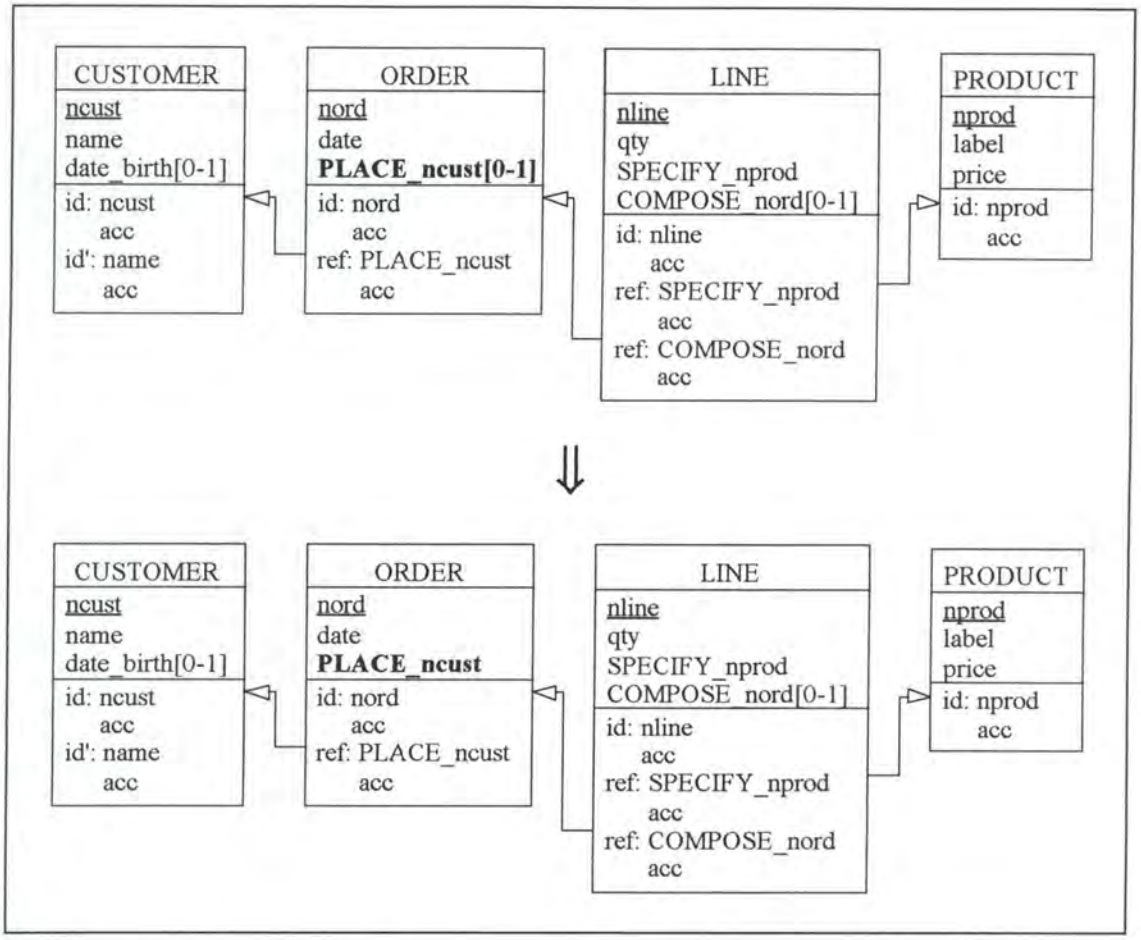


Figure A1 - 48 : Augmenting the minimum cardinality of a role to 1 in an 0-1/0-N relationship-type on the logical level

4.2.2.2.2. SQL Description & Data

```
delete from ORDER
  where PLACE_ncust is null
alter table ORDER
  alter PLACE_ncust  not null  constraint O_PLACE_ncust;
```

We loose the rows of table ORDER for which no CUSTOMER was specified.

4.2.2.2.3. Program Extracts

Similar remarks as for the case 4.2.2.1.1.3. (see page A1-61) can be formulated here.

5. MODIFICATIONS OF THE ATTRIBUTES

5.1. MODIFICATIONS WHICH AUGMENT THE SEMANTICS

5.1.1. Add_optional_attribute

Let us suppose we want to add an optional attribute firstname to CUSTOMER.

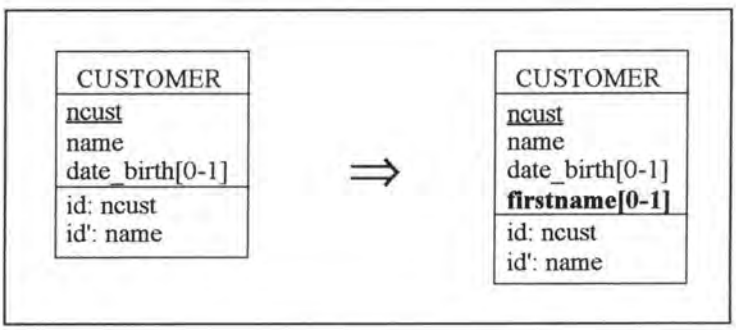


Figure A1 - 49 : Adding an optional attribute on the conceptual level

5.1.1.1. Logical Schema

We add an optional column firstname to the relation CUSTOMER.

5.1.1.2. SQL Description & Data

```
alter table CUSTOMER
  add firstname    char(15);
```

Note that all the rows of CUSTOMER have a null value for column firstname.

CUSTOMER			
ncust	name	(date_birth)	**(firstname)**
A101	Bootsma H.	12/07/1969	**null**
D308	Ford H.	null	**null**
B234	Peiffer M.	22/06/1917	**null**
A958	Huntington G.	31/01/1969	**null**
D365	McGaw J.	29/02/1980	**null**
:	:	:	:

Figure A1 - 50 : Table CUSTOMER after having added column firstname

5.1.1.3. Program Extracts

Let us consider the following program extract:

```
type dat: ...

var cust: STRING[4];
    name: STRING[12];
    date_birth: dat;

  :
exec SQL
      select *
        into :cust, :name, :date_birth
        from CUSTOMER
        where ncust = 'A101';
end exec
  :
```

It can either be modified as follows:

```
type dat: ...

var cust : STRING[4];
    name: STRING[12];
    date_birth: dat;
    firstname: STRING[15];

  :
exec SQL
      select *
        into :cust, :name, :date_birth, :firstname
        from CUSTOMER
        where ncust = 'A101';
end exec
  :
```

or as follows:

```
type dat: ...

var cust : STRING[4];
    name: STRING[12];
    date_birth: dat;
```

```
  :
exec SQL
      select ncust, name, date_birth
        into :cust, :name, :date_birth
        from CUSTOMER
        where ncust = 'A101';
end exec
  :
```

We have to change the application programs by adding variables (as illustrated here above) or by assigning an output field for firstname in the user interfaces.

5.1.2. Add_mandatory_attribute

Let us suppose we want to add as well a mandatory attribute telephone to CUSTOMER.

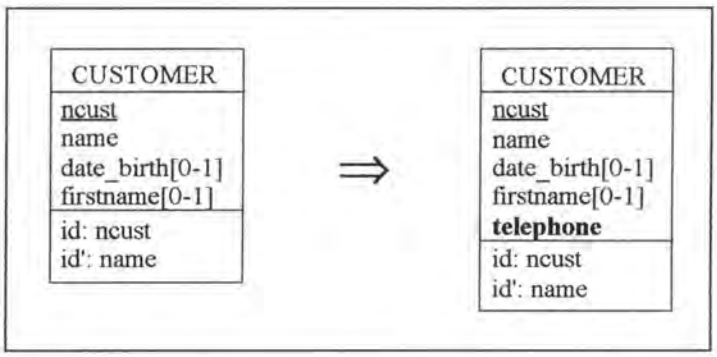


Figure A1 - 51 : Adding a mandatory attribute on the conceptual level

5.1.2.1. Logical Schema

We add a column telephone to the relation CUSTOMER.

5.1.2.2. SQL Description & Data

```
alter table CUSTOMER
  add telephone  varchar(20)  default '0' not null  constraint C_telephone;
```

In order to keep all the data of CUSTOMER, we place the default value '0' in column telephone for each row of the table CUSTOMER:

CUSTOMER				
ncust	name	(date_birth)	(firstname)	telephone
A101	Bootsma H.	12/07/1969	null	0
D308	Ford H.	null	null	0
B234	Peiffer M.	22/06/1917	null	0
A958	Huntington G.	31/01/1969	null	0
D365	McGaw J.	29/02/1980	null	0
:	:	:	:	:

Figure A1 - 52 : Table CUSTOMER after having added column telephone too

5.1.2.3. Program Extracts

Similar remarks can be formulated as for the case add_optional_attribute (see page A1-68).

5.1.3. Make_attr_optional

Precondition:
As SQL-RDB does not allow optional attributes as primary key, the attribute that should be made optional must not be a primary key.

As we do not always know the price of a PRODUCT, we want to make it optional.

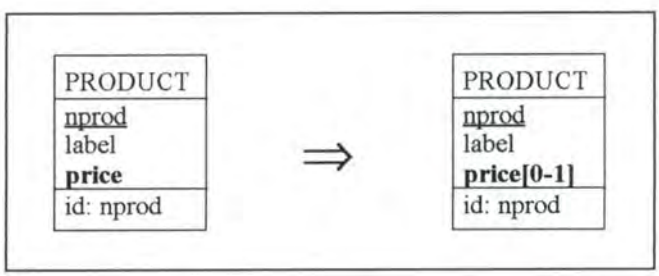


Figure A1 - 53 : Making an attribute optional on the conceptual level

5.1.3.1. Logical Schema

We make column price optional in relation PRODUCT.

5.1.3.2. SQL Description & Data

```
alter table PRODUCT
  drop constraint P_price;
```

Note that the data will not be changed as we only make the column price optional.

5.1.3.3. Program Extracts

Let us consider the following program extract:

```
var price: INTEGER;

  :
exec SQL
      select price
        into :price
        from PRODUCT
        where nprod = 'AA110'
end exec;
if SQLCODE = 0
then write (price);
else write ('The product ''AA110'' does not exist.');
  :
```

As PRODUCT 'AA110' has now not necessarily a price anymore, we have to change the previous program extract as follows:

```
var price, null_indicator: INTEGER;

  :
exec SQL
      select price
        into :price:null_indicator
        from PRODUCT
        where nprod = 'AA110'
end exec;
if SQLCODE = 0          (* if such a row has been found *)
then if null_indicator = 0    (* if the price is known *)
     then write (price)
     else write ('The price of PRODUCT ''AA110'' is unknown.')
else write ('The product ''AA110'' does not exist.');
  :
```

As we can see in the previous program extract, some variables and tests checking the null value of column price must be added.

5.1.4. Extend_domain_attribute

Precondition:
The attribute whose domain should be modified must not be an identifier. This is due to SQL-RDB which allows modifications only on columns, on which no constraints (primary, unique and foreign key) apply. In addition, the attribute cannot be of the type date.

Let us suppose we want to extend the domain of column label in entity-type PRODUCT from char(20) to char(25).

5.1.4.1. Logical Schema

We extend the domain of column label in table PRODUCT from char(20) to char(25).

5.1.4.2. SQL Description & Data

```
alter table PRODUCT
  drop constraint P_label,
  alter label char(25) not null constraint P_label;
```

Note:
If label were optional, we would have to do the following operation:

```
alter table PRODUCT
  alter label char(25);
```

No modifications are made on the data.

5.1.4.3. Program Extracts

In the application programs the variables, the procedure arguments and the user interface output fields referencing column label of table PRODUCT must be adapted accordingly. For example:

```
var label: STRING[20];

  :
exec SQL
      select label
        into :label
        from PRODUCT
        where nprod = 'SW226'
end exec;
  :
```

```
var label: STRING[25];

  :
exec SQL
      select label
        into :label
        from PRODUCT
        where nprod = 'SW226'
end exec;
  :
```

5.1.5. Change_type_int_char

Precondition:
The attribute must not be an identifier (neither a primary nor a unique key). This is due to SQL-RDB which allows modifications only on columns, on which no constraints (primary, unique and foreign key) apply.

Let us suppose that we want to change the type of attribute price to char(11) in entity-type PRODUCT of our case study.

5.1.5.1. Logical Schema

We have to change the type of column price to char(11) in relation PRODUCT.

5.1.5.2. SQL Description & Data

```
var i: INTEGER;
    s: STRING[11];

exec SQL
      (* an intermediate column is created *)
      alter table PRODUCT
        add p  integer;
      (* the data of column price is copied into that column *)
      update PRODUCT
        set p = price;
      (* the old column price is replaced by the new one *)
      alter table PRODUCT
        drop constraint P_price,    (* we remove the mandatory feature from
                                       the old column price *)
        drop price,
        add price  char(11)  default '0'  not null constraint P_price;
      (* the data is converted and copied into the new column price *)
      declare c cursor for
        select p
          from PRODUCT
        for update of price in PRODUCT;
      open c;
      fetch c into :i;
end exec;
while SQLCODE = 0       (* the last item has not yet been treated *)
do begin
      s := f_int_char(i);
      exec SQL
           update PRODUCT
             set price = :s
             where current of c;
           fetch c into :i;
      end exec;
   end;
exec SQL
      close c;
      (* the intermediate column is dropped *)
      alter table PRODUCT
        drop p;
end exec;
```

f_int_char:
This function converts an integer into a string.

No data is lost, but we have to note that the values of column price are converted.

5.1.5.3. Program Extracts

In the application programs the user interface output fields, the variables, the procedure arguments and sometimes the constants referencing column price of PRODUCT must be adapted accordingly. For example:

```
var price: INTEGER;

  :
exec SQL
```

```
      select price
        into :price
        from PRODUCT
        where nprod = 'SW226'
end exec;
  :
```

```
var price: STRING[11];

  :
exec SQL
      select price
        into :price
        from PRODUCT
        where nprod = 'SW226'
end exec;
  :
```

5.1.6. Change_type_float_char

This modification is similar to the previous one (see page A1-72), except that we use function f_float_char instead of f_int_char. Function f_float_char converts a float into a string.

5.1.7. Change_type_date_char

This modification is similar to change_type_int_char (see page A1-72), except that we use function f_date_char instead of f_int_char. Function f_date_char converts a date into a string.

5.1.8. Change_type_date_int

This modification is similar to change_type_int_char (see page A1-72), except that we use function f_date_int instead of f_int_char. Function f_date_int converts a date into an integer.

5.1.9. Change_type_int_float

This modification is similar to change_type_int_char (see page A1-72), except that we use function f_int_float instead of f_int_char. Function f_int_float converts an integer into a float.

5.1.10. Change_type_date_float

This modification is similar to change_type_int_char (see page A1-72), except that we use function f_date_float instead of f_int_char. Function f_date_float converts a date into a float.

5.2. MODIFICATIONS WHICH DECREASE THE SEMANTICS

5.2.1. Remove_optional_attribute

Precondition:
The attribute that has to be removed must not be the last one of the entity-type.

Let us suppose that we want to remove the attribute date_birth from the entity-type CUSTOMER of our case study.

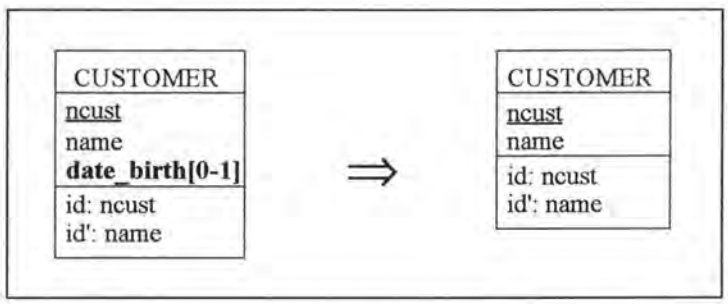


Figure A1 - 54 : Removing an optional attribute on the conceptual level

5.2.1.1. Logical Schema

We remove the column date_birth from relation CUSTOMER.

5.2.1.2. SQL Description & Data

```
alter table CUSTOMER
  drop date_birth;
```

Note:
In case date_birth were a unique key, we would have to use the following operation:

```
alter table CUSTOMER
  drop constraint idCUS3,      (* we remove the unique key
                                  feature *)
  drop date_birth;
```

All the data of column date_birth will be lost:

CUSTOMER	
ncust	name
A101	Bootsma H.
D308	Ford H.
B234	Peiffer M.
A958	Huntington G.
D365	McGaw J.
:	:

Figure A1 - 55 : Table CUSTOMER when column date_birth is removed

5.2.1.3. Program Extracts

The first SELECT query of our case study (see page 4-7) does not make sense anymore and must therefore be dropped. The JOIN query (see page 4-8) may be modified as follows:

```
select name, nord
  from CUSTOMER, ORDER
  where ncust = PLACE_ncust.
```

Note that this query gives now the following result:

name	nord
Huntington G.	E386
Hasselhoff S.	F285
Osborn M.	F842
Peiffer M.	E345
Huntington G.	G222
McGaw J.	F902
Hasselhoff S.	E583
Bootsma H.	F676

As we have already said, it is often not sufficient to delete or modify the select queries only. The application programs in which they appear must also be reviewed: certain variables may be dropped and certain user interfaces may be adapted.

5.2.2. Remove_mandatory_attribute

Precondition:
The attribute which should be removed must not be a primary key and must not be the last attribute of the entity-type.

Let us imagine we want to remove attribute date from entity-type ORDER of our case study.

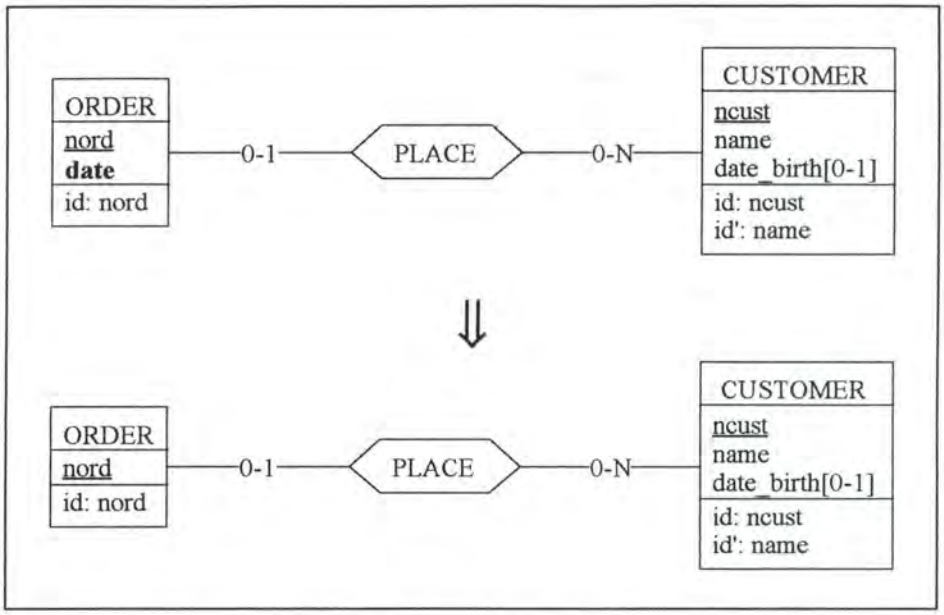


Figure A1 - 56 : Removing a mandatory attribute on the conceptual level

5.2.2.1. Logical Schema

We remove the column date from relation ORDER of our case study.

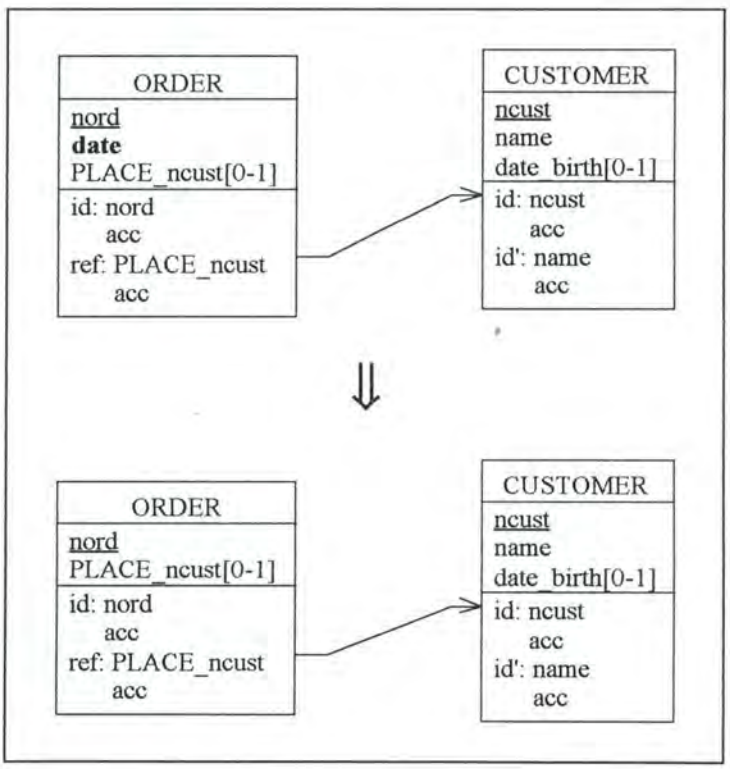


Figure A1 - 57 : Removing a mandatory attribute on the logical level

5.2.2.2. SQL Description & Data

```
alter table ORDER
  drop constraint O_date,      (* we remove the mandatory feature from
                                  column date *)
  drop date;
```

Note:
In case date were a unique key, we would have to use the following operation:

```
alter table ORDER
  drop constraint idORD2,      (* we remove the unique key
                                  feature *)
  drop constraint O_date,      (* we remove the mandatory
                                  feature from column date *)
  drop date;
```

All the data of column date will be lost:

ORDER	
nord	(PLACE_ncust)
E386	A958
F285	B472
G274	null
F842	C395
:	:

ORDER.PLACE_ncust in CUSTOMER.ncust

Figure A1 - 58 : Table ORDER when column date is removed

5.2.2.3. Program Extracts

Every select query referencing date must either be dropped or modified in a similar way as for the case remove_optional_attribute (see page A1-76). The remark on the application programs also applies here.

5.2.3. Make_attr_mandatory

We have to distinguish whether the attribute which we want to make mandatory is a unique key or not. We will treat first the case in which the attribute is not a unique key.

5.2.3.1. The attribute is not a unique key

Let us suppose we want to make date_birth mandatory in CUSTOMER.

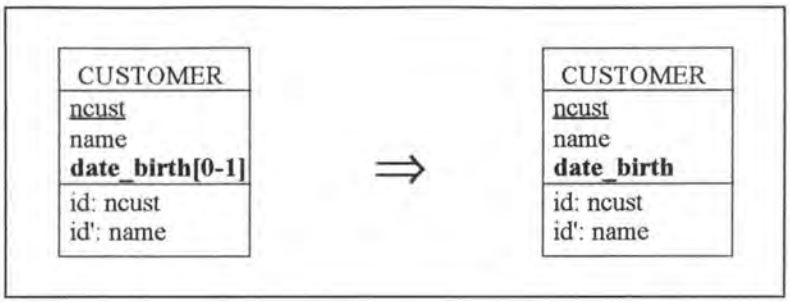


Figure A1 - 59 : Making a non-key attribute mandatory on the conceptual level

5.2.3.1.1. Logical Schema

We make date_birth mandatory in relation CUSTOMER.

5.2.3.1.2. SQL Description & Data

```
update CUSTOMER
  set date_birth = 00/00/0000
  where date_birth is null;
alter table CUSTOMER
  alter date_birth not null constraint C_date_birth;
```

This way of implementing the modification involves no loss of data as we replace the null values of column date_birth by a default value:

CUSTOMER		
ncust	name	date_birth
A101	Bootsma H.	12/07/1969
D308	Ford H.	**00/00/0000**
B234	Peiffer M.	22/06/1917
A958	Huntington G.	31/01/1969
D365	McGaw J.	29/02/1980
B472	Hasselhoff S.	**00/00/0000**
C385	Casci G.	**00/00/0000**
A590	Nutbush M.	09/06/1969
B253	Whopper J.	**00/00/0000**
C395	Osborn M.	28/11/1972

Figure A1 - 60 : Table CUSTOMER when the null values of column date_birth are replaced by a default value

We thus have the choice whether to remove or not the rows of table CUSTOMER with the default value in column date_birth. If we want to remove those rows, we can use the following operation:

```
delete
  from CUSTOMER
  where date_birth = 00/00/0000
```

We then still have to decide what should happen to the ORDERs PLACEd by the CUSTOMER B472. Note that the only CUSTOMER who has PLACEd ORDERs is CUSTOMER B472. We have two choices:

A. Set PLACE_ncust to null for the ORDERs PLACEd by the CUSTOMER B472.

ORDER		
nord	(PLACE_ncust)	date
E386	A958	02/01/1995
F285	**null**	12/03/1994
G274	null	15/07/1993
F842	C395	31/12/1994
E345	B234	05/01/1995
G222	A958	23/05/1994
F902	D365	16/09/1994
E583	**null**	12/01/1995
F676	A101	26/02/1993
G809	null	23/05/1994

ORDER.PLACE_ncust in CUSTOMER:ncust

Figure A1 - 61 : Table ORDER when certain PLACE_ncust values are set to null

B. Delete also the ORDERs PLACEd by the CUSTOMER B472.

ORDER		
nord	(PLACE_ncust)	date
E386	A958	02/01/1995
G274	null	15/07/1993
F842	C395	31/12/1994
E345	B234	05/01/1995
G222	A958	23/05/1994
F902	D365	16/09/1994
F676	A101	26/02/1993
G809	null	23/05/1994

ORDER.PLACE_ncust in CUSTOMER:ncust

Figure A1 - 62 : Table ORDER when certain rows are deleted

If we have decided to delete also the ORDERs, we have finally to decide what should happen to the LINEs which COMPOSE the ORDERs E583 and F285. Here again we have the two same choices:

B1. Set COMPOSE_nord to null for the LINEs associated to the ORDERs that have been removed:

LINE			
nline	(COMPOSE_nord)	qty	SPECIFY_nprod
AB1234	E386	1000	AA110
GH2345	null	1518	CA510
RT3456	**null**	345	AA110
ZU4567	G274	2536	BE072
ER5678	null	4587	EG880
NM6789	G274	5558	WN592
OP7890	G274	5458	RK560
JK0987	F842	5473	SW226
TZ9876	E386	623	LS906
KJ8765	null	4587	SG953
WQ7654	F902	6325	BY907
XY6543	null	9658	BY907
DS5432	**null**	5698	EG880
BV1357	F842	7458	AB099
IO2468	G809	4125	AB099

LINE.COMPOSE_nord in ORDER.nord
LINE.SPECIFY_nprod in PRODUCT.nprod

Figure A1 - 63 : Table LINE where certain values for column COMPOSE_nord are set to null

B2. Delete also the LINEs associated to the ORDERs that have been removed:

LINE			
nline	(COMPOSE_nord)	qty	SPECIFY_nprod
AB1234	E386	1000	AA110
GH2345	null	1518	CA510
ZU4567	G274	2536	BE072
ER5678	null	4587	EG880
NM6789	G274	5558	WN592
OP7890	G274	5458	RK560
JK0987	F842	5473	SW226
TZ9876	E386	623	LS906
KJ8765	null	4587	SG953
WQ7654	F902	6325	BY907
XY6543	null	9658	BY907
BV1357	F842	7458	AB099
IO2468	G809	4125	AB099

LINE.COMPOSE_nord in ORDER.nord
LINE.SPECIFY_nprod in PRODUCT.nprod

Figure A1 - 64 : Table LINE where certain rows are deleted

5.2.3.1.3. Program Extracts

Select queries referencing the null value of column date_birth could be modified as follows:

```
select ncust
  from CUSTOMER
  where date_birth is null
```

```
select ncust
  from CUSTOMER
  where date_birth = 00/00/0000
```

in case we have not dropped the data and should be dropped else.
Note that all the application programs in which such queries appear must also be reviewed. For example, the tests on the null value of date_birth must be changed either by testing the default value or by simply removing them.

5.2.3.2. The attribute is a unique key

We want to make mandatory the attribute label in entity-type FACTORY.

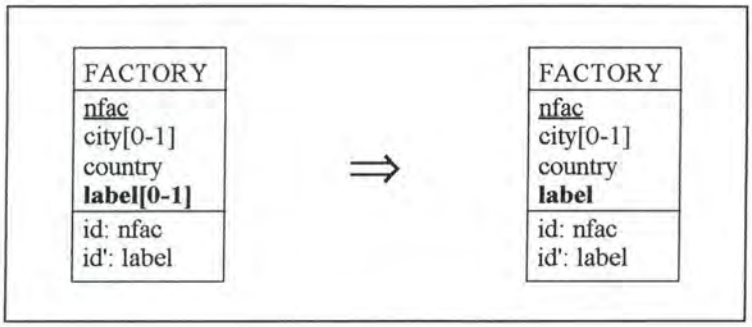


Figure A1 - 65 : Making an attribute which is a unique key mandatory on the conceptual level

5.2.3.2.1. Logical Schema

We make the column label in relation FACTORY mandatory.

5.2.3.2.2. SQL Description & Data

```
(* we cannot use here a default value because of the unique key feature of
   column label *)
delete from FACTORY
  where label is null;
alter table FACTORY          (* we can only alter a column on which no
                                constraints apply *)
  drop constraint idFAC2,     (* we remove the unique key constraint *)
  alter label not null constraint F_label,
  add constraint unique(label) constraint idFAC2;
```

All the rows which had a null value for column label in table FACTORY are lost.

5.2.3.2.3. Program Extracts

All the application programs containing queries referencing the null value of column label must be reviewed in a similar way as in the previous case (see page A1-83).

5.2.4. Restrict_domain_attribute

Precondition:
The attribute whose domain should be modified must not be an identifier. This is due to SQL-RDB which allows modifications only on columns, on which no constraints (primary, unique and foreign key) apply. In addition, the attribute cannot be of the type date.

Let us suppose we want to restrict the domain of attribute label from char(20) to char(15) in entity-type PRODUCT.

5.2.4.1. Logical Schema

We restrict the domain of column label in table PRODUCT from char(20) to char(15).

5.2.4.2. SQL Description & Data

```
alter table PRODUCT
  drop constraint P_label,
  alter label char(15) not null constraint P_label;
```

Note:
If label were optional, we would have to do the following operation:

```
alter table PRODUCT
  alter label char(15);
```

SQL-RDB truncates values already stored in the database that exceed the capacity of the new data type, but only when it retieves those values. (The values are not truncated in the database, however, until they are updated. If you only retrieve data, therefore, you can change the data type back to the original, and SQL again retrieves the entire original value.) [RDB91, page 7-48]

5.2.4.3. Program Extracts

In the application programs the variables, the procedure arguments and the user interface output fields referencing column label of PRODUCT have to be adapted accordingly. For example:

```
var label: STRING[20];

  :
exec SQL
      select label
        into :label
        from PRODUCT
        where nprod = 'SW226'
end exec;
  :
```

⇓

```
var label: STRING[15];

  :
exec SQL
      select label
        into :label
        from PRODUCT
        where nprod = 'SW226'
end exec;
  :
```

5.2.5. Change_type_char_int

This modification is similar to change_type_int_char (see page A1-72), except that we use function f_char_int instead of f_int_char. Function f_char_int converts a string into an integer. Depending on the implementation of function f_char_int, we could loose data.

5.2.6. Change_type_float_int

This modification is similar to change_type_int_char (see page A1-72), except that we use function f_float_int instead of f_int_char. Function f_float_int converts a float into an integer. Depending on the implementation of function f_float_int, we could loose data.

5.2.7. Change_type_char_float

This modification is similar to change_type_int_char (see page A1-72), except that we use function f_char_float instead of f_int_char. Function f_char_float converts a string into a float. Depending on the implementation of function f_char_float, we could loose data.

5.2.8. Change_type_char_date

This modification is similar to change_type_int_char (see page A1-72), except that we use function f_char_date instead of f_int_char. Function f_char_date converts a string into a date. Depending on the implementation of function f_char_date, we could loose data.

5.2.9. Change_type_int_date

This modification is similar to change_type_int_char (see page A1-72), except that we use function f_int_date instead of f_int_char. Function f_int_date converts an integer into a date. Depending on the implementation of function f_int_date, we could loose data.

5.2.10. Change_type_float_date

This modification is similar to change_type_int_char (see page A1-72), except that we use function f_float_date instead of f_int_char. Function f_float_date converts a float into a date. Depending on the implementation of function f_float_date, we could loose data.

5.3. MODIFICATIONS WHICH PRESERVE THE SEMANTICS

5.3.1. Rename_optional_attribute

We have to distinguish whether the optional attribute is a unique key or not. We will first treat the case where the attribute is not a unique key.

5.3.1.1. The attribute is not a unique key

We want to rename date_birth into d_birth in entity-type CUSTOMER.

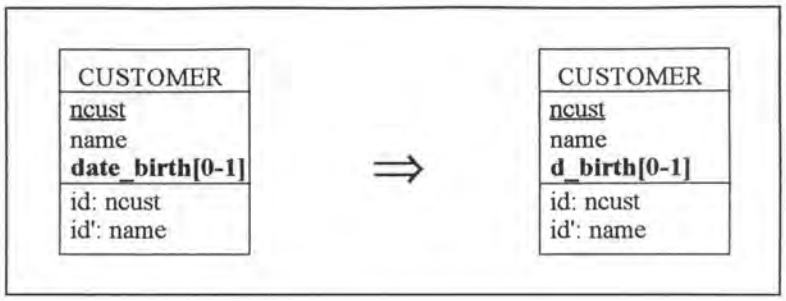


Figure A1 - 66 : Renaming an optional attribute on the conceptual level

5.3.1.1.1. Logical Schema

We rename column date_birth into d_birth in relation CUSTOMER.

5.3.1.1.2. SQL Description & Data

```
alter table CUSTOMER
  add d_birth    date;
update CUSTOMER
  set d_birth = date_birth;
alter table CUSTOMER
  drop date_birth;
```

Note:
In order to avoid copying a whole column, we can create a view. But as the view name must be unique among all view and table names in the schema, we would have to change all the select queries referencing that table. We thus prefer the first approach though it is rather slowly to be executed.

No data is lost as the data is copied from one column into another.

5.3.1.1.3. Program Extracts

The first SELECT query of our case study (see page 4-7) must be modified as follows:

```
select *
. from CUSTOMER
  where d_birth = 09/06/1969.
```

The JOIN query (see page 4-8) must also be changed and becomes then:

```
select name, nord
  from CUSTOMER
  where (ncust = PLACE_ncust) and (d_birth < 01/01/1977).
```

In fact, in every query referencing date_birth, it must be replaced by d_birth. Sometimes it might be good to rename also certain labels of the user interface output fields and certain variables accordingly.

5.3.1.2. The attribute is a unique key

Let us consider the entity-type FACTORY where we want to rename attribute label into name.

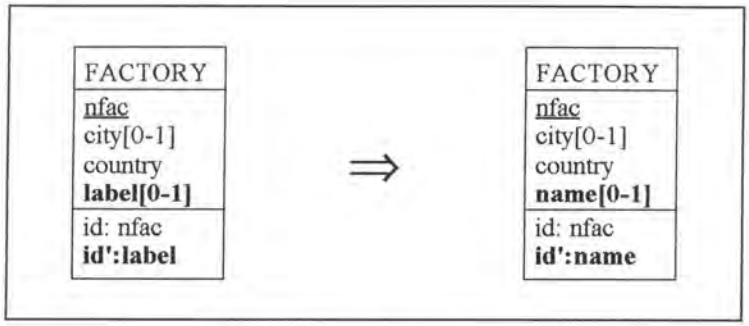


Figure A1 - 67 : Renaming an optional attribute on the conceptual level

5.3.1.2.1. Logical Schema

We rename label into name in relation FACTORY.

5.3.1.2.2. SQL Description & Data

```
alter table FACTORY
  add name char(20);
update FACTORY
  set name = label;
alter table FACTORY
  drop constraint idFAC2,     (* we remove the old unique key feature *)
  drop label,
  add constraint unique(name) constraint idFAC2;
```

Note:
In order to avoid copying a whole column, we can create a view. But as the view name must be unique among all view and table names in the schema, we would have to change all the select queries referencing that table. We thus prefer the first approach though it is rather slowly to be executed.

No data is lost as the data is only copied from one column into another.

5.3.1.2.3. Program Extracts

Similar remarks as in the previous case can be formulated (see page A1-87).

5.3.2. Rename_mandatory_attribute

Precondition:
In order to avoid having also to rename the foreign keys, we require that the attribute which should be renamed must not be a primary key.

We distinguish whether the attribute is a unique key or not.

5.3.2.1. The attribute is not a unique key

Let us suppose that we want to change label into description in PRODUCT.

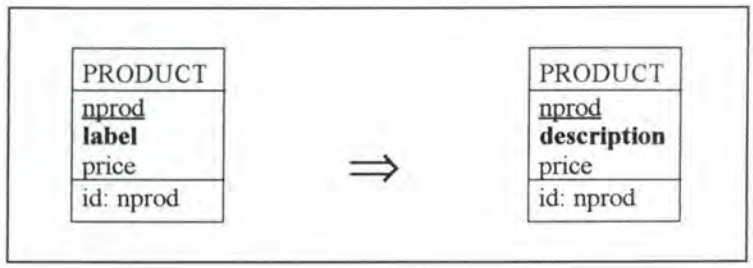


Figure A1 - 68 : Renaming a mandatory attribute on the conceptual level

5.3.2.1.1. Logical Schema

We rename the column label into description in relation PRODUCT.

5.3.2.1.2. SQL Description & Data

```
alter table PRODUCT
  add description  char(20)  default 'X' not null constraint P_description;
update PRODUCT
  set description = label;
alter table PRODUCT
  drop constraint P_label,    (* we remove the mandatory feature from
                                 column label *)
  drop label;
```

Note:
In order to avoid copying a whole column, we can create a view. But as the view name must be unique among all view and table names in the schema, we would have to change all the select queries referencing that table. We thus prefer the first approach though it is rather slowly to be executed.

No data is lost as we only copy the data from one column into another.

5.3.2.1.3. Program Extracts

In every select query referencing column label of table PRODUCT, label must be replaced by description. Sometimes it might be good to rename also certain labels of the user interface output fields and certain variables accordingly.

5.3.2.2. The attribute is a unique key

Let us suppose we want to rename the attribute name in CUSTOMER into surname.

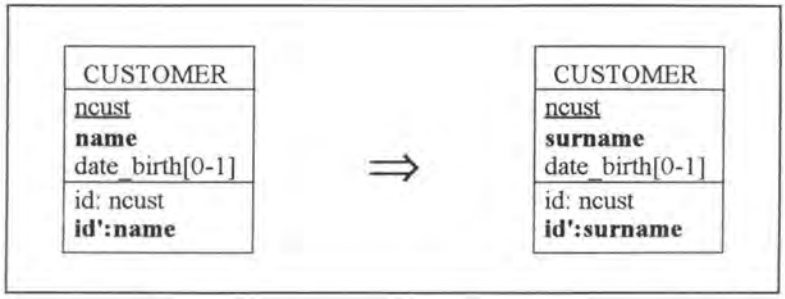


Figure A1 - 69 : Renaming a mandatory attribute on the conceptual level

5.3.2.2.1. Logical Schema

We have to rename column name into surname in relation CUSTOMER.

5.3.2.2.2. SQL Description & Data

```
alter table CUSTOMER
  add surname char(12) default 'X' not null constraint C_surname;
update CUSTOMER
  set surname = name;
alter table CUSTOMER
  drop constraint idCUS2,      (* we remove the old unique key feature *)
  drop constraint C_name,      (* we remove the mandatory feature from
                                  column name *)
  drop name,
  add constraint unique(surname) constraint idCUS2;
```

Note:
In order to avoid copying a whole column, we can create a view. But as the view name must be unique among all view and table names in the schema, we would have to change all the select queries referencing that table. We thus prefer the first approach though it is rather slowly to be executed.

No data is lost as the data is only copied from one column into another.

5.3.2.2.3. Program Extracts

The JOIN query (see page 4-8) must be modified as follows:

```
select surname, nord
  from CUSTOMER, ORDER
  where (ncust = PLACE_ncust) and (date_birth < 01/01/1977)
```

In fact, in every select query referencing name of table CUSTOMER, it must be replaced by surname. Sometimes it might be good to rename also certain labels of the user interface output fields and certain variables accordingly.

6. MODIFICATIONS OF THE IDENTIFIER

6.1. MODIFICATIONS WHICH AUGMENT THE SEMANTICS

6.1.1. Remove_unique_feature

Let us suppose we want to remove the uniqueness constraint from attribute name in entity-type CUSTOMER.

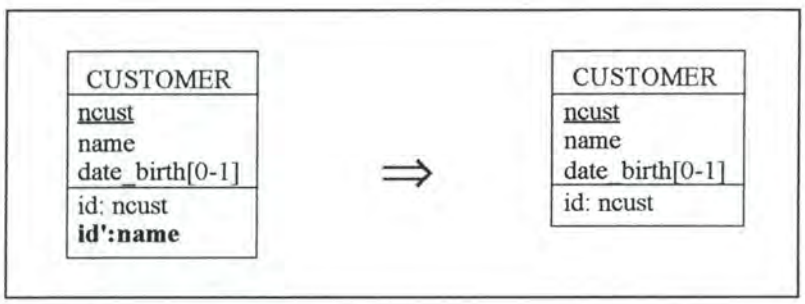


Figure A1 - 70 : Removing a unique key feature on the conceptual level

6.1.1.1. Logical Schema

We remove the uniqueness constraint from column name in relation CUSTOMER.

6.1.1.2. SQL Description & Data

```
alter table CUSTOMER
  drop constraint idCUS2;
```

No changes are made on the data.

6.1.1.3. Program Extracts

Certain program extracts must be changed. For example:

```
var cust: STRING[4];

exec SQL
      select ncust
        into :cust
        from CUSTOMER
        where name = 'Nutbush M.'
end exec;
```

```
if SQLCODE = 0
then write(cust);
```

```
var cust: STRING[4];

exec SQL
      declare c cursor for
        select ncust
          from CUSTOMER
          where name = 'Nutbush M.';
      open c;
      fetch c into :cust;
end exec;
while SQLCODE = 0 (* the last item has not yet been treated *)
do begin
      write(cust);
      exec SQL
            fetch c into :cust
      end exec;
   end;
exec SQL
      close c;
end exec;
```

As we can see in the previous program extracts, simple test conditions must be transformed into loops.

6.2. MODIFICATIONS WHICH DECREASE THE SEMANTICS

6.2.1. Add_unique_feature

Let us suppose we want to make attribute label a unique key of PRODUCT.

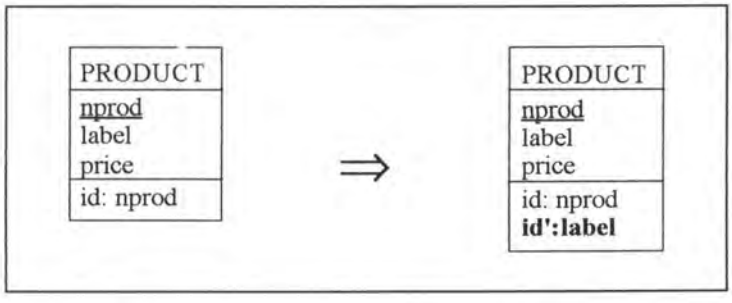


Figure A1 - 71 : Adding a unique key feature on the conceptual level

6.2.1.1. Logical Schema

We have to add the unique key feature to column label in relation PRODUCT.

6.2.1.2. SQL Description & Data

```
(* we add the unique key feature to column label *)
alter table PRODUCT
  add constraint unique (label) constraint idPRO2;
```

In our case study example, no data would have been lost, generally however this modification involves loss of data.

6.2.1.3. Program Extracts

Certain program extracts can be simplified. For example:

```
var prod: STRING[5];

  :
exec SQL
      declare c cursor for
        select nprod
          from PRODUCT
          where label = 'christmas tree';
      open c;
      fetch c into :prod;
end exec;
while SQLCODE = 0 (* the last item has not yet been treated *)
do begin
      write(prod);
      exec SQL
            fetch c into :prod
      end exec;
   end;
exec SQL
      close c;
end exec;
  :
```

```
var prod: STRING[5]

  :
exec SQL
      select nprod
        into :prod
        from PRODUCT
        where label = 'christmas tree';
end exec;
if SQLCODE = 0
then write(prod);
  :
```

As we can see in the previous program extracts, loops may be transformed into simple test conditions.

6.3. MODIFICATIONS WHICH PRESERVE THE SEMANTICS

6.3.1. Switch_PK_unique

This modification is used to transform a primary key into a unique key and vice versa. The user has the choice whether to specify a unique key or not. If he does not specify any unique key, then a technical identifier is created as primary key.

Precondition:
If a unique key is specified, then it must not be optional, as SQL-RDB does not allow optional attributes as primary key.

The structure of the modification switch_PK_unique is represented in Figure A1-72.

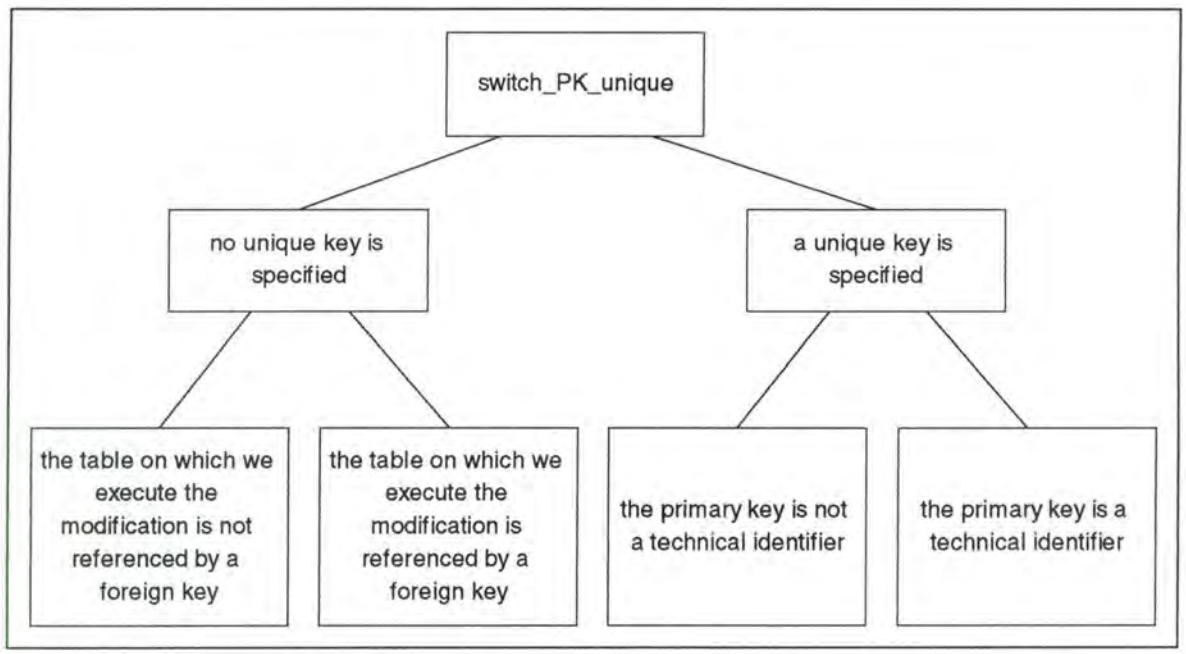


Figure A1 - 72 : Structure of the modification switch_PK_unique

For each one of the four basic cases we will reconsider Figure A1-72 indicating in bold the current position.

6.3.1.1. There is no unique key specified

Let us suppose we have the entity-type ADDRESS and that we want to transform the primary key nadd into a unique key.

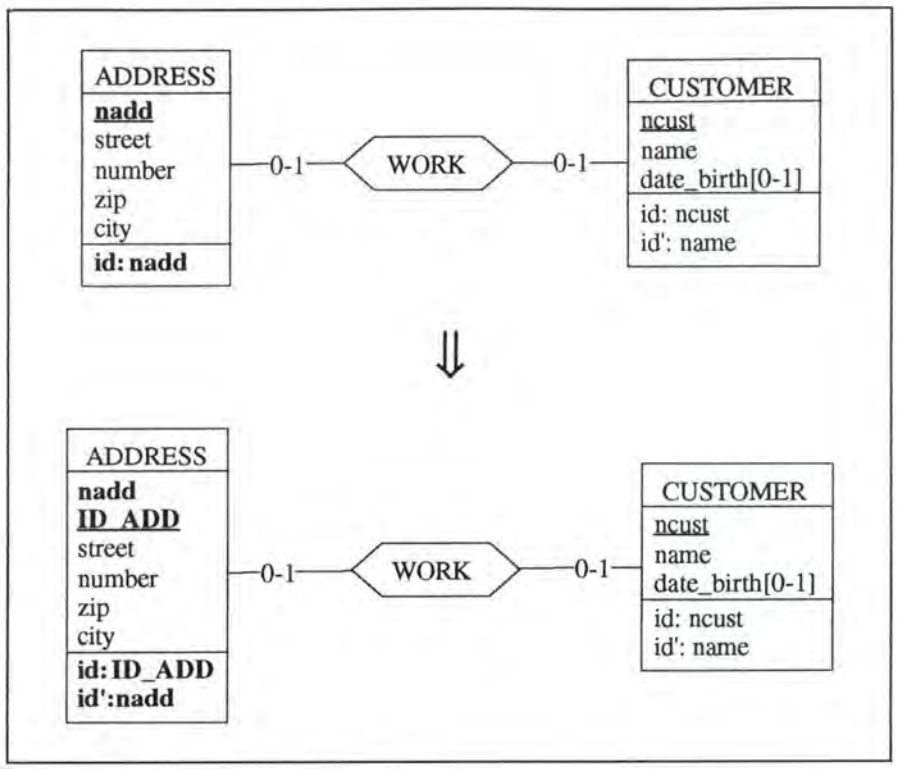
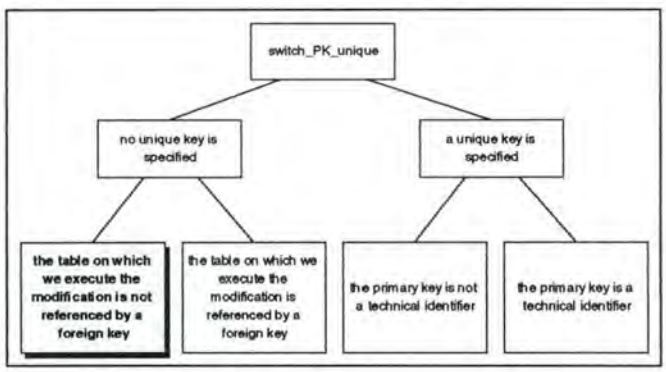


Figure A1 - 73 : Transforming a primary key into a unique key when no unique key is specified, on the conceptual level

Two different cases must be considered:

- WORK is implemented by a foreign key in ADDRESS
- WORK is implemented by a foreign key in CUSTOMER

6.3.1.1.1. WORK is implemented by a foreign key in ADDRESS



6.3.1.1.1.1. Logical Schema

We transform the primary key into a unique key, create a technical identifier and promote it to a primary key:

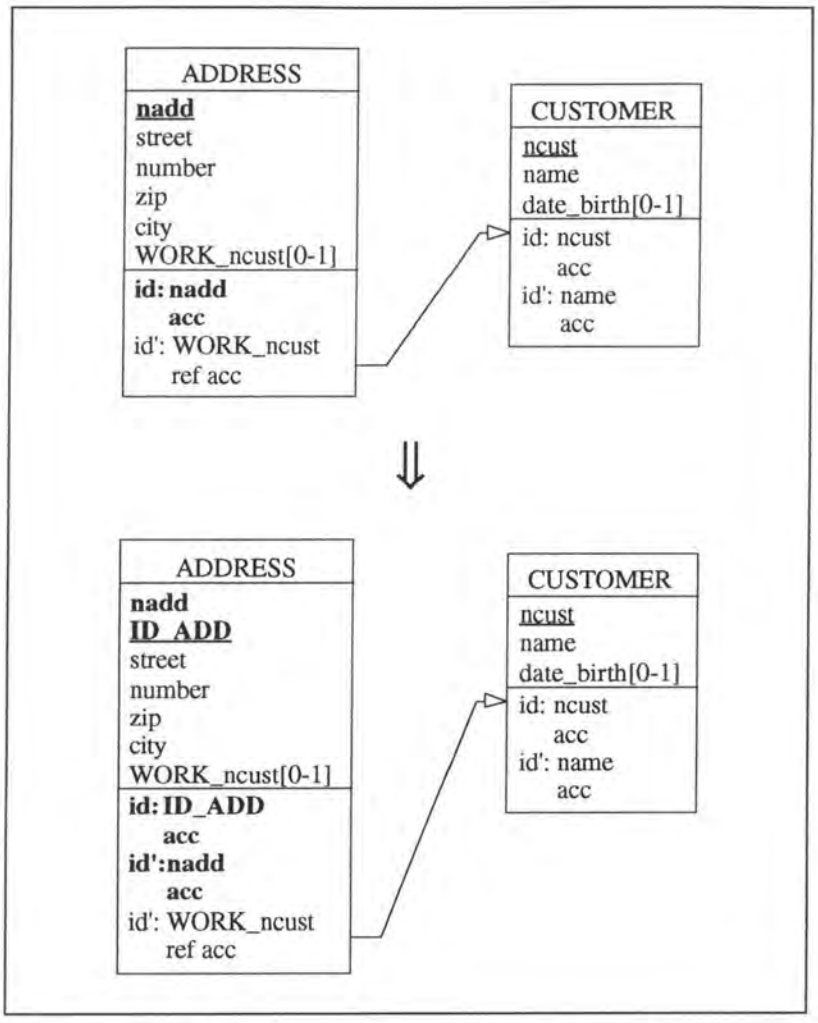


Figure A1 - 74 : Transforming a non referenced primary key into a unique key when no unique key is specified, on the logical level

6.3.1.1.1.2. SQL Description & Data

```
var i: INTEGER;

exec SQL
      (* we create the technical identifier column *)
      alter table ADDRESS
        add ID_ADD smallint  default 0 not null constraint A_ID_ADD;
      (* we assign identifying values to that column *)
      declare c cursor for
        select ID_ADD
          from ADDRESS
        for update of ID_ADD in ADDRESS;
      open c;
      fetch c;
end exec;
i:= 1;
while SQLCODE = 0       (* the last item has not yet been treated *)
do begin
      exec SQL
            update ADDRESS
              set ID_ADD = :i
              where current of c;
```

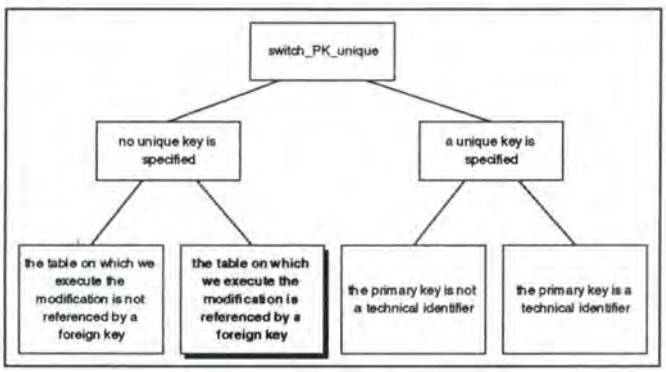
```
            fetch c;
      end exec;
      i:= i+1;
   end;
exec SQL
      close c;
      (* we operate the 'real switch' *)
      alter table ADDRESS
        drop constraint idADD1      (* we drop the old primary key
                                                    constraint *),
        add constraint primary key(ID_ADD) constraint idADD1,
        add constraint unique(nadd) constraint idADD3;
end exec;
```

No data is lost as we only manipulate identifying features and add a technical identifier.

6.3.1.1.1.3. Program Extracts

There is no impact on the application programs.

6.3.1.1.2. WORK is implemented by a foreign key in CUSTOMER



6.3.1.1.2.1. Logical Schema

We transform the primary key nadd into a unique key, create a technical identifier which we promote to a primary key and change also the foreign key referencing table ADDRESS.

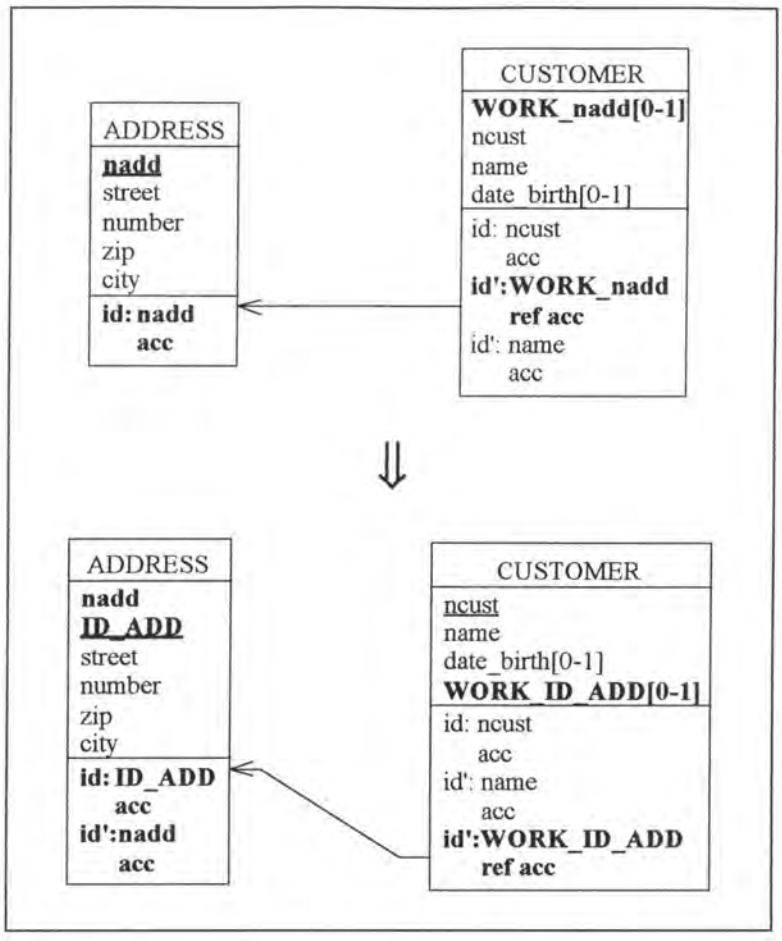


Figure A1 - 75 : Transforming a referenced primary key into a unique key when no unique key is specified, on the logical level

6.3.1.1.2.2. SQL Description & Data

```
var add, i, idADD: INTEGER;

exec SQL
      (* we create the technical identifier column *)
      alter table ADDRESS
        add ID_ADD smallint  default 0 not null constraint A_ID_ADD;
      (* we assign identifying values to that column *)
      declare c1 cursor for
        select ID_ADD
          from ADDRESS
        for update of ID_ADD in ADDRESS;
      open c1;
      fetch c1;
end exec;
i:= 1;
while SQLCODE = 0       (* the last item has not yet been treated *)
do begin
      exec SQL
            update ADDRESS
              set ID_ADD = :i
              where current of c1;
            fetch c1;
      end exec;
      i:= i+1;
```

```
   end;
exec SQL
      close c1;
      (* we replace the foreign key column representing the
         relationship-type WORK *)
      alter table CUSTOMER
        add WORK_ID_ADD smallint,
        drop constraint ADD1;
      declare c2 cursor for
        select WORK_nadd
          from CUSTOMER
          where WORK_nadd is not null
        for update of WORK_ID_ADD;
      open c2;
      fetch c2 into:add
end exec;
while SQLCODE = 0  (* the last item has not yet been treated *)
do begin
      exec SQL
           select ID_ADD
             into :idADD
             from ADDRESS
             where nadd = :add;
           update CUSTOMER
             set WORK_ID_ADD = :idADD
             where current of c2;
           fetch c2 into :add;
      end exec;
   end;
exec SQL
      (* we operate the 'real switch' and adapt the foreign key
         constraints *)
      alter table ADDRESS
        drop constraint idADD1,     (* we drop the old primary key
                                       constraint *)
        add constraint primary key(ID_ADD)constraint idADD1,
        add constraint unique(nadd) constraint idADD2;
      alter table CUSTOMER
        drop constraint idCUS3,
        add constraint foreign key(WORK_ID_ADD) references ADDRESS
                                                  constraint ADD1,
        add constraint unique(WORK_ID_ADD) constraint idCUS3,
        drop WORK_nadd;
      close c2;
end exec;
```

No data is lost as we only manipulate identifying features, add a technical identifier and 'copy' the data representing relationship-type WORK from column WORK_nadd into column WORK_ID_ADD.

6.3.1.1.2.3. Program Extracts

We have to review all the application programs referencing the foreign key representing relationship-type WORK. For example:

-
```
var name: STRING[12];

exec SQL
      select name
        into :name
        from CUSTOMER
        where WORK_nadd = 102;
end exec;
if SQLCODE = 0
then write(name);
```

⇓

```
var name: STRING[12];

exec SQL
     select name
       into :name
       from CUSTOMER
       where WORK_ID_ADD = 52;
           (* Let us suppose ADDRESS has the value 52 for ID_ADD
              if it had the value 102 for nadd *)
end exec;
if SQLCODE = 0
then write(name);
```

-
```
select street, city
  from ADDRESS
  where nadd in ( select WORK_nadd
                    from CUSTOMER
                    where name like '%Dupont%' )
```

⇓

```
select street, city
  from ADDRESS
  where ID_ADD in ( select WORK_ID_ADD
                          from CUSTOMER
                          where name like '%Dupont%' )
```

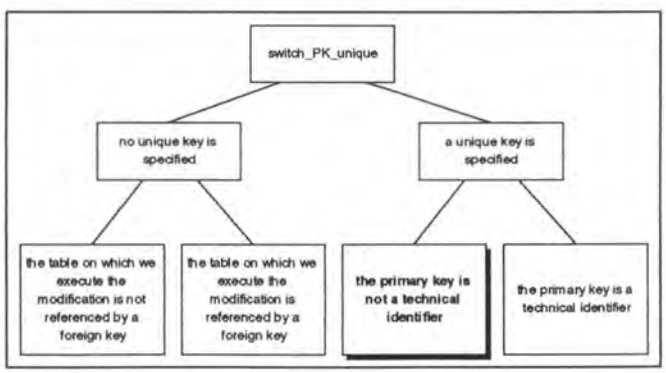
6.3.1.2. The unique key is specified

We have here to distinguish again two cases:

- The primary key is not a technical one
- The primary key is a technical one

For each of these two cases we would have to distinguish again whether the table on which we execute the modification is referenced by a foreign key or not. As these subcases would not bring any new ideas, we will not distinguish them.

6.3.1.2.1. The primary key is not a technical one



Let us suppose we want to replace the primary key ncust of CUSTOMER by the unique key name.

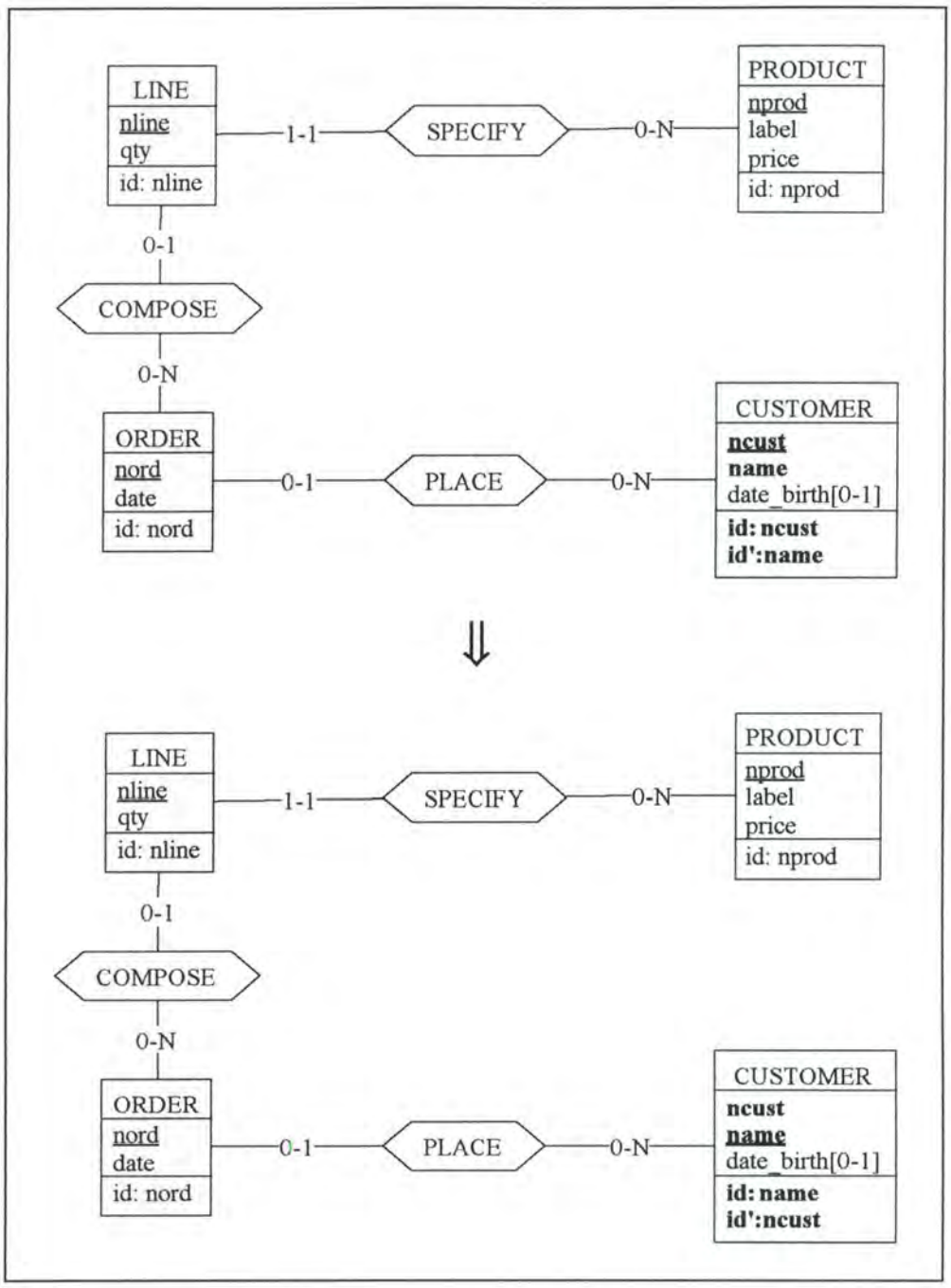


Figure A1 - 76 : Replacing a non technical primary key by a unique key on the conceptual level

6.3.1.2.1.1. Logical Schema

We transform the primary key ncust into a unique key, make the unique key name a primary key and change also the foreign key referencing table ORDER.

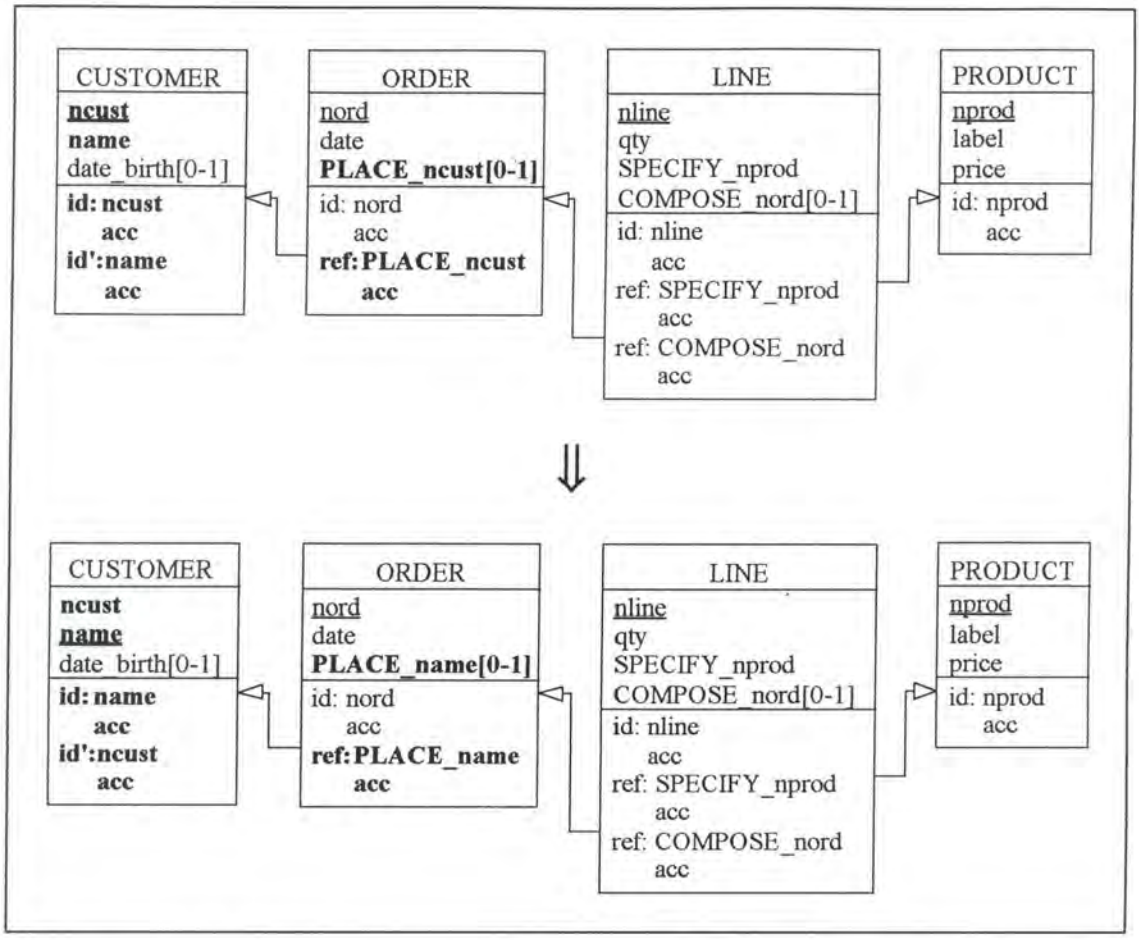


Figure A1 - 77 : Replacing a non technical primary key by a unique key on the logical level

6.3.1.2.1.2. SQL Description & Data

```
var cust: STRING[4];
    name: STRING[12];

exec SQL
      (* we replace the foreign key column representing the
         relationship-type PLACE *)
      alter table ORDER
        add PLACE_name char(12),
        drop constraint CUS1;
      declare c cursor for
        select PLACE_ncust
          from ORDER
          where PLACE_ncust is not null
        for update of PLACE_name;
      open c;
      fetch c into:cust
end exec;
while SQLCODE = 0  (* the last item has not yet been treated *)
do begin
      exec SQL
           select name
             into :name
             from CUSTOMER
             where ncust = :cust;
           update ORDER
```

```
            set PLACE_name = :name
            where current of c;
          fetch c into :cust;
      end exec;
   end;
exec SQL
      (* we operate the 'real switch' and adapt the foreign key
         constraints *)
      alter table CUSTOMER
        drop constraint idCUS1,     (* we drop the old primary key
                                       constraint *)
        drop constraint idCUS2,     (* we drop the old unique key
                                       constraint *)
        add constraint primary key(name) constraint idCUS1,
        add constraint unique(ncust) constraint idCUS2;
      alter table ORDER
        add constraint foreign key(PLACE_name) references CUSTOMER
                                               constraint CUS1,
        drop PLACE_ncust;
      close c;
end exec;
```

No data is lost as we only manipulate identifying features and 'copy' the data representing relationship-type PLACE from column PLACE_ncust into column PLACE_name.

6.3.1.2.1.3. Program Extracts

The second SELECT query of our case study (see page 4-7) must be modified as follows:

```
select *
  from CUSTOMER
  where name in (select PLACE_name
                   from ORDER
                   where nord in (select COMPOSE_nord
                                  from LINE
                                  where SPECIFY_nprod = 'AA110'))
```

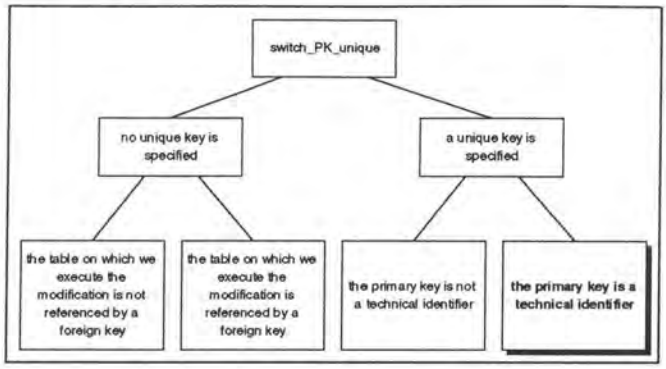
The JOIN query (see page 4-8) becomes:

```
select name, nord
  from CUSTOMER, ORDER
  where (name = PLACE_name) and (date_birth < 01/01/1977).
```

In fact, every program extract referencing PLACE_ncust must be reviewed. For example, let us suppose that the user must give the number of a CUSTOMER (ncust) in order to get his ORDERs. As relationship-type PLACE is now represented by the foreign key PLACE_**name**, either the user has to indicate the name of the CUSTOMER or we have to insert the following query before executing the remaining of the program:

```
select name
  from CUSTOMER
  where ncust = <the number given by the user>
```

6.3.1.2.2. The primary key is a technical one



Let us suppose we have the entity-type ADDRESS where the primary key is a technical identifier and nadd is a unique key. We want now to make nadd a primary key and drop the technical identifier ID_ADD.

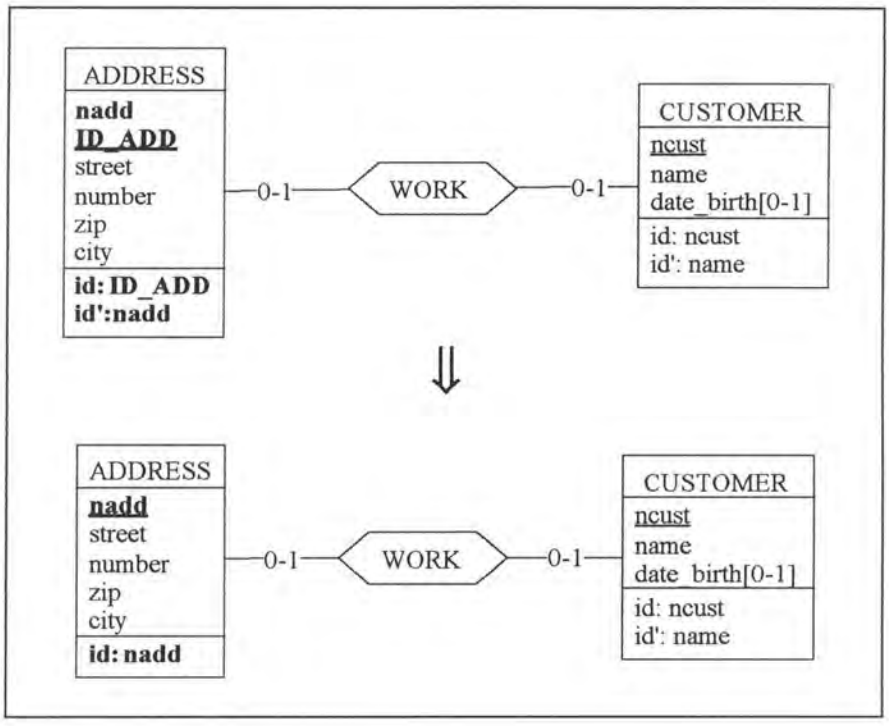


Figure A1 - 78 : Replacing a technical primary key by a unique key on the conceptual level

6.3.1.2.2.1. Logical Schema

In order to simplify, we only consider the case where the relationship-type WORK has been implemented by the foreign key in relation ADDRESS. We have to make nadd a primary key and drop ID_ADD.

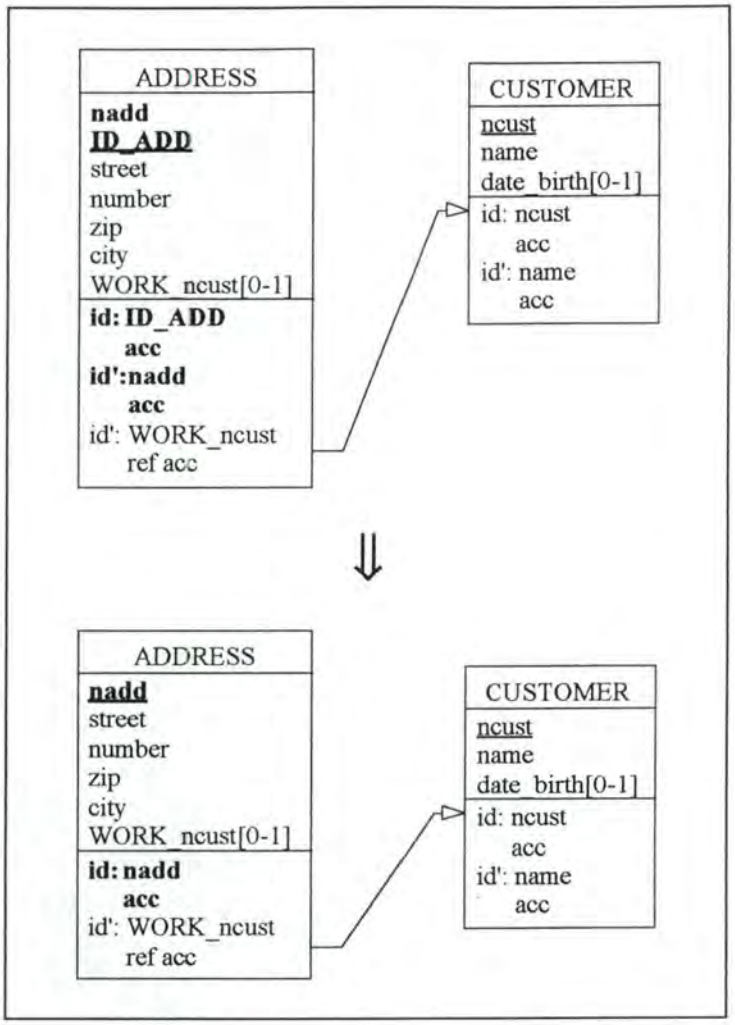


Figure A1 - 79 : Replacing a technical primary key by a unique key on the logical level

6.3.1.2.2.2. SQL Description & Data

```
alter table ADDRESS
  drop constraint idADD1,      (* we drop the old primary key
                                  constraint *)
  drop constraint idADD2,      (* we drop the old unique key
                                  constraint *)
  drop constraint A_ID_ADD,    (* we remove the mandatory feature from
                                  column ID_ADD *)
  add constraint primary key(nadd) constraint idADD1,
  drop ID_ADD;
```

No data is lost as we do not consider the information included in column ID_ADD as semantical data.

6.3.1.2.2.3. Program Extracts

There is no impact on the application programs as ID_ADD is a technical construct and is thus not referenced by any query.

Appendix 2:

Study of the Modifications: General Approach

TABLE OF CONTENTS

Appendix 2 :Study of the Modifications: General Approach

TABLE OF FIGURES

Appendix 2 :Study of the Modifications: General Approach

1. INTRODUCTION

After having studied the modifications on a case study, we will analyse them in general. We have here again to study the modifications of the conceptual level and their impacts on the logical level, on the SQL database structure, on the data and on the application programs. This is illustrated by Figure A2-1.

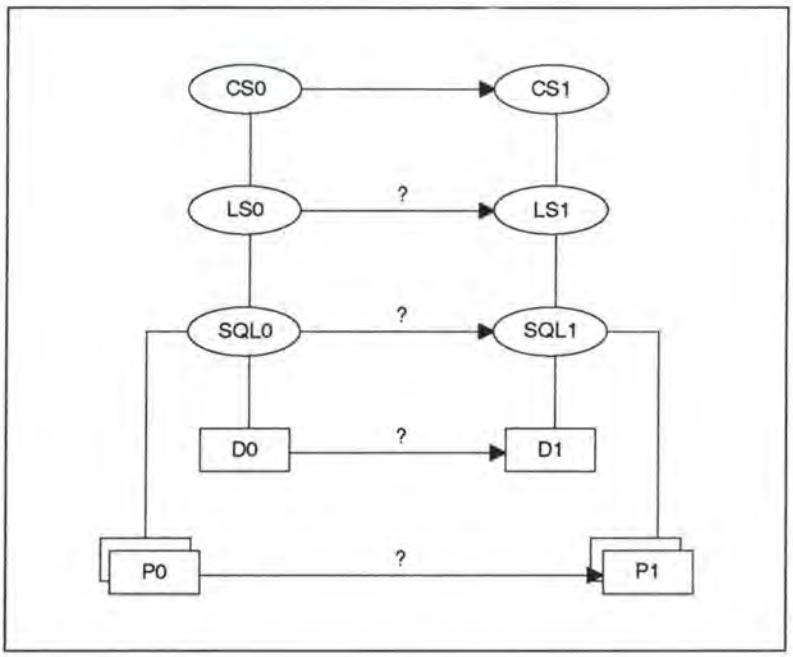


Figure A2 - 1 : Representation of the database evolution problem

If the conceptual schema CS0 has been changed, the logical schema LS0 and the SQL description SQL0 must be changed accordingly. Data D0 is no longer valid and has to be converted into data D1. Finally, the applications P0 must be partly rewritten in order to comply with the new data structures described in SQL1.[HAI94a]

As shown in the third chapter, the modifications are classified according to the objects on which they apply on one hand and, on the other hand, according to their nature: augmenting, decreasing or preserving semantics (see page 3-4).

In order not to loose the overview of this appendix, we will give once more the typology of the modifications. We will indicate in bold those modifications which are redundant with those detailed in chapter 5.

Modifications of the entity-types which:

augment the semantics: add_entity-type

decrease the semantics: remove_entity-type

preserve the semantics: **rename_entity-type**

Modifications of the relationship-types which:

augment the semantics:
add_1-1/0-1_rel-type
add_0-1/0-1_rel-type
add_1-1/0-N_rel-type
add_0-1/0-N_rel-type

decrease the semantics:
remove_1-1/0-1_rel-type
remove_0-1/0-1_rel-type
remove_1-1/0-N_rel-type
remove_0-1/0-N_rel-type

preserve the semantics:
rename_1-1/0-1_rel-type
rename_0-1/0-1_rel-type
rename_1-1/0-N_rel-type
rename_0-1/0-N_rel-type

Modifications of the roles which:

augment the semantics:
augment_max_card
decrease_min_card

decrease the semantics:
decrease_max_card
augment_min_card

Modifications of the attributes which:

augment the semantics:
add_optional_attribute
add_mandatory_attribute
make_attr_optional
extend_domain_attribute
change_type_int_char
change_type_float_char
change_type_date_char
change_type_date_int
change_type_int_float
change_type_date_float

decrease the semantics:
remove_optional_attribute
remove_mandatory_attribute
make_attr_mandatory
restrict_domain_attribute
change_type_char_int
change_type_float_int

change_type_char_float
change_type_char_date
change_type_int_date
change_type_float_date

preserve the semantics: rename_optional_attribute
rename_mandatory_attribute

Modifications of the identifiers which:

augment the semantics: remove_unique_feature

decrease the semantics: add_unique_feature

preserve the semantics: **switch_PK_unique**

For each object, we thus distinguish three types of modifications: those augmenting, decreasing and preserving the semantics. Within each of these three parts, we develop for each modification its impact on the Logical Schema, on the SQL Description & Data and on the Program Extracts.

2. STUDY OF THE MODIFICATIONS: GENERAL APPROACH

2.1. MODIFICATIONS OF THE ENTITY-TYPES

2.1.1. Modifications which Augment the Semantics

2.1.1.1. Add_entity-type[1]

Note:
Each entity-type must have at least one attribute and must have a primary key.

Let us suppose we want to add the following entity-type E1:

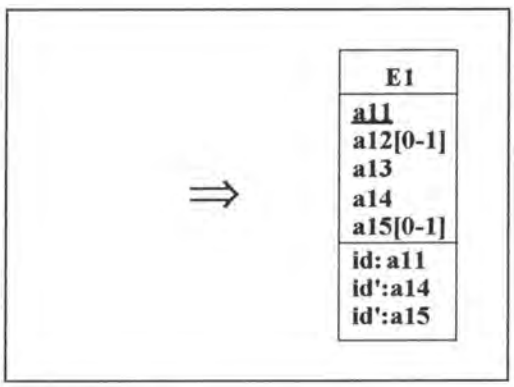


Figure A2 - 2 : Adding an entity-type on the conceptual level

2.1.1.1.1. Logical Schema

On the logical level, we have to add the corresponding relation.

2.1.1.1.2. SQL Description & Data

```
create table E1
      ( a11 <type>  not null  constraint E1_a11,
        a12 <type>,
        a13 <type>  not null  constraint E1_a13,
        a14 <type>  not null  constraint E1_a14,
        a15 <type>,
        primary key (a11) constraint idE1_#[2],
```

[1]Normally we would have to add here the following precondition: 'The name of the entity-type that should be added must not yet exist.' As such preconditions are trivial, we will not indicate them anymore.

```
unique (a14) constraint idE1_#,
unique (a15) constraint idE1_# )
```

There is no effect on the existing data.

2.1.1.1.3. Program Extracts

There is no change on the existing application programs. The documentation must however be updated. As the changes of the documentation are necessary for all the modifications, we will not indicate them anymore in this appendix.

2.1.2. Modifications which Decrease the Semantics

2.1.2.1. Remove_entity-type

Precondition:
The entity-type that has to be removed must not be connected to any relationship-type.

Let us suppose we want to remove the entity-type E1.

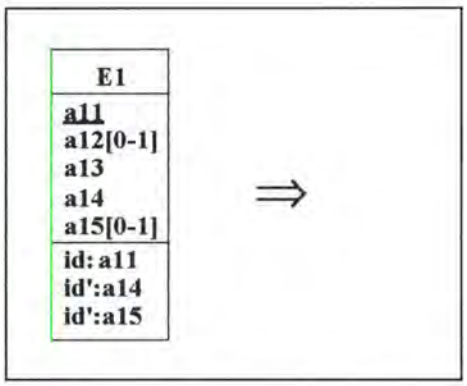


Figure A2 - 3 : Removing an entity-type on the conceptual level

2.1.2.1.1. Logical Schema

In the logical schema we remove the relation E1 with its columns and all their constraints.

2.1.2.1.2. SQL Description & Data

```
drop table E1 cascade;
```

Note that all the data included in table E1 will be lost.

[2] As it is difficult to indicate the proper number for each constraint, we will use the symbol #.

2.1.2.1.3. Program Extracts

Most of the select queries which reference table E1 are invalid. For example:

- ```
  select ...
    from E1
    where ...
  ```

- ```
  select ...
    from E2
    where a21 in ( select a13
                     from E1
                     where ... )
  ```

- ```
  select ...
    from E2
    where ...
  UNION
  select ...
    from E1
    where ...
  ```

- ```
  select a11, a21, a31
    from E1, E2, E3
    where (a22 = a12) and (a32 = 100*a22)
  ```

The way these queries are modified depends on each one individually. The application programs in which they appear must also be reviewed. We cannot describe a general method how to deal with these application programs as each one of them must be treated individually, depending on their context. A CASE tool offering this modification should only indicate the concerned program extracts and give sometimes hints about the changes to be done. For example, certain variables should be deleted and certain user interfaces should be reviewed.

2.1.3. Modifications which Preserve the Semantics

2.1.3.1. Rename_entity-type

Let us suppose we want to rename the entity-type E into E1.

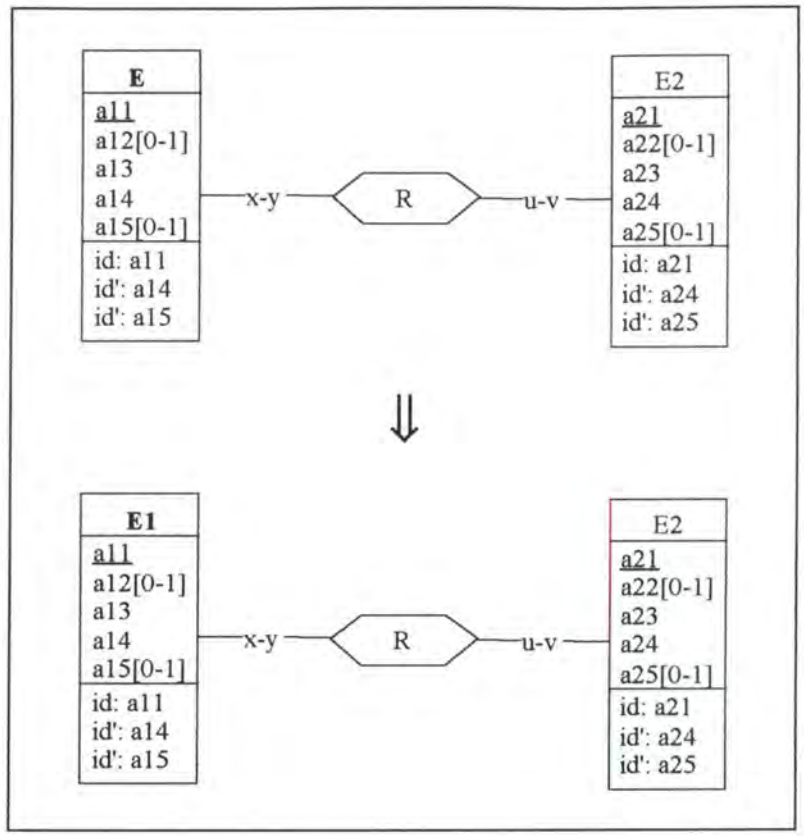


Figure A2 - 4: Renaming an entity-type on the conceptual level

2.1.3.1.1. Logical Schema

In the logical schema, we have to change the name of the corresponding relation. Due to the parametrical cardinalities, different cases are possible. In Figure A2-5 however (and only there), we will only illustrate the two basic ones:

- R is represented by a foreign key in E
- R is represented by a foreign key in E2

For each of these two cases, we will only consider the situation where relationship-type R has one 0-N role. The other cases would be similar, except that we would have to express identifying features.

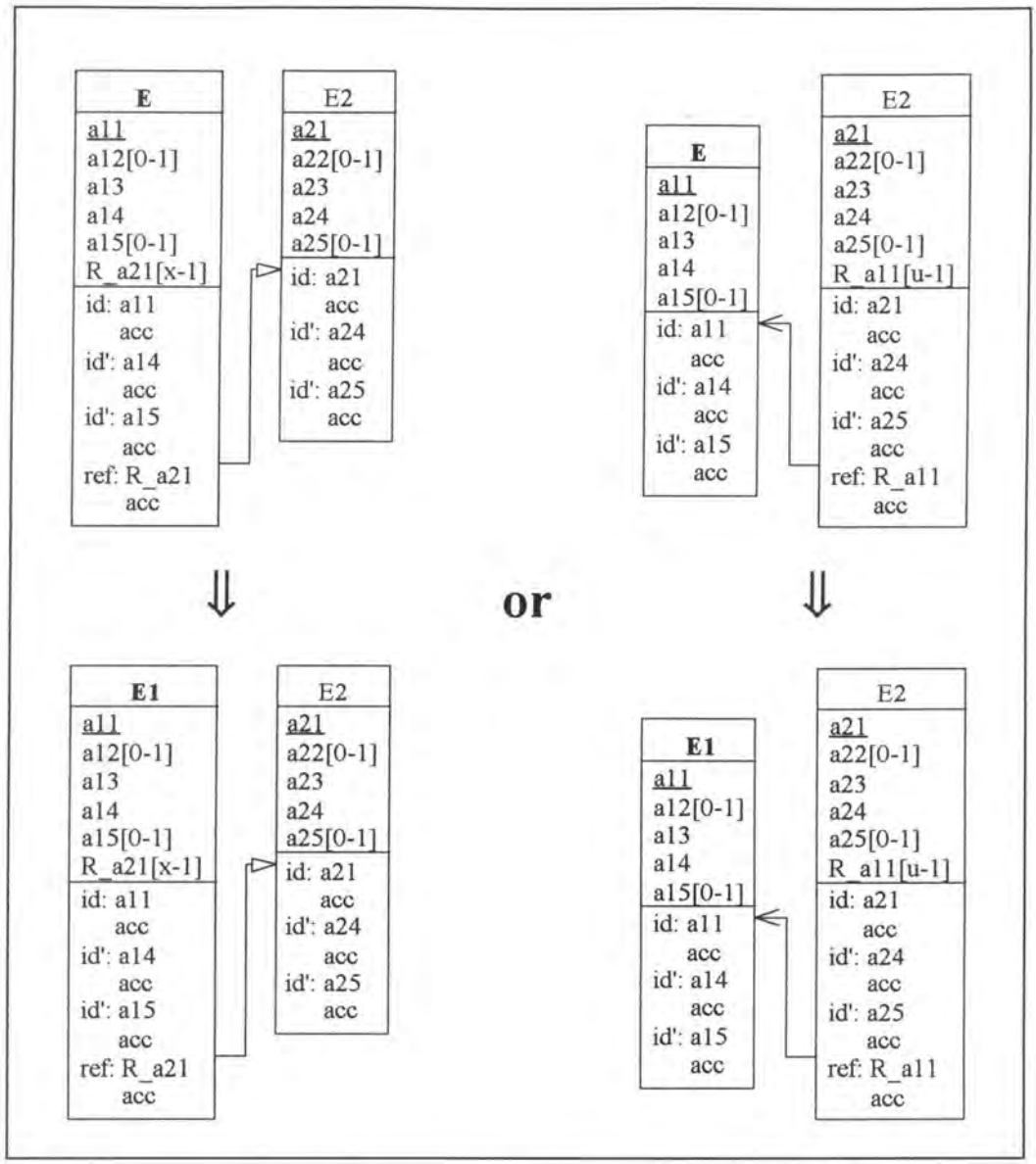


Figure A2 - 5 : Renaming an entity-type on the logical level

2.1.3.1.2. SQL Description & Data

In some SQL languages there may be a 'rename table' command. The modification would then become:

```
alter table E
  rename table E1 on cascade;
```

In SQL-RDB however, no such command exists and we have therefore to create a new table and to copy the data into it.

```
exec SQL
      (* we create table E1 *)
      create table E1
            ( a11 <type>  not null  constraint E1_a11,
```

```
              a12 <type>,
              a13 <type>  not null  constraint E1_a13,
              a14 <type>  not null  constraint E1_a14,
              a15 <type>,
              primary key (a11) constraint idE1_#[3],
              unique (a14) constraint idE1_#,
              unique (a15) constraint idE1_# )
end exec;
(* we create the foreign keys in E1 *)
for each of the relationship-types R connected to E
do if R is represented by a foreign key in E
   then begin
           if x = 0
           then exec SQL
                      alter table E1
                        add R_a21  <type>;
                end exec
           else exec SQL
                      alter table E1
                        add R_a21  <type>  default <value>  not null
                                                  constraint E1_R_a21;
                end exec;
           if v = 1
           then exec SQL
                      alter table E1
                        add constraint unique (R_a21)  constraint idE1_#;
                end exec;
           exec SQL
                alter table E1
                  add constraint foreign key (R_a21) references E2
                                                  constraint E2_#;
           end exec;
        end;
exec SQL
     (* we insert the data of E into E1 *)
     insert into E1
       select *
       from E
end exec;
(* we redirect to E1 the foreign keys referencing E *)
for each of the relationship-types R connected to E
do if R is represented by a foreign key in E2
   then exec SQL
              alter table E2
                drop constraint E_#;  (* we remove the old foreign key
                                         feature *)
                add constraint foreign key (R_a11) references E1
                                                  constraint E1_#,
        end exec;

(* For each view defined on table E, we have to redefine it on E1. In
future we will not consider views anymore as they do not correspond to ER
objects. *)

drop table E cascade;
```

No data is lost as the data is just moved from one table into another.

Notes:

- This operation in SQL-RDB is often a very slow one as we have to copy a whole table. We thus recommend to create a view E1 which includes only the table E. This could be realized by the following command:

[3]As it is difficult to indicate the proper number for each constraint, we will use the symbol #.

```
create view E1
  as select *
       from E
```

- Other SQL languages, such as DB2, offer another possibility to implement the modification: giving a synonym to the entity-type (that has to be renamed) instead of renaming it properly. This alternative could be realised by the following SQL command:

```
create synonym E1
  for E
```

Note that in both cases the original table is however not renamed.

2.1.3.1.3. Program Extracts

- In all the select queries referencing E, we have to rename it with E1.
  For example:

```
select ...
  from E
  where ...
```

```
select ...
  from E1
  where ...
```

- In the following example, we have to rename E not only in the 'from' clause, but also in the 'where' clause:

```
select ...
  from E, E2
  where E.a = E2.a
```

```
select ...
  from E1, E2
  where E1.a = E2.a
```

- In addition to the select queries, we have also to review the application programs in which they appear. For instance, we must rename certain variables and/or some fields or headings in the user interfaces. Finally, let us note that the documentation should also be updated.

2.2. MODIFICATIONS OF THE RELATIONSHIP-TYPES

2.2.1. Modifications which Augment the Semantics

2.2.1.1. Add_1-1/0-1_rel-type

Let us suppose we want now to link entity-type E1 to the entity-type E2 by a 1-1/0-1 relationship-type R.

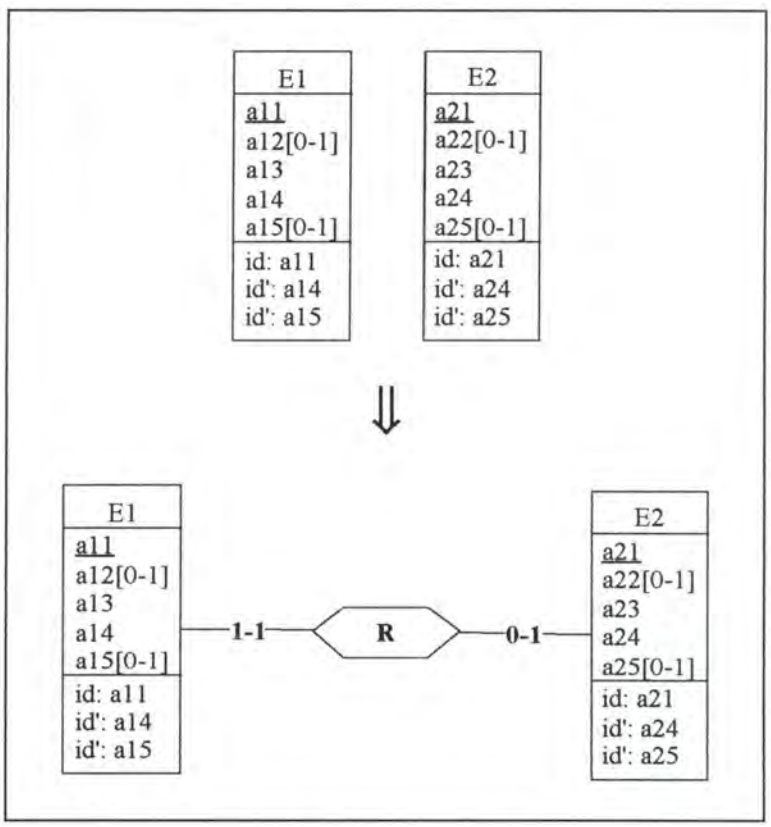


Figure A2 - 6 : Adding a 1-1/0-1 relationship-type on the conceptual level

2.2.1.1.1. Logical Schema

In the logical schema we add the primary key a21 of E2 to E1 as a foreign and a candidate key.

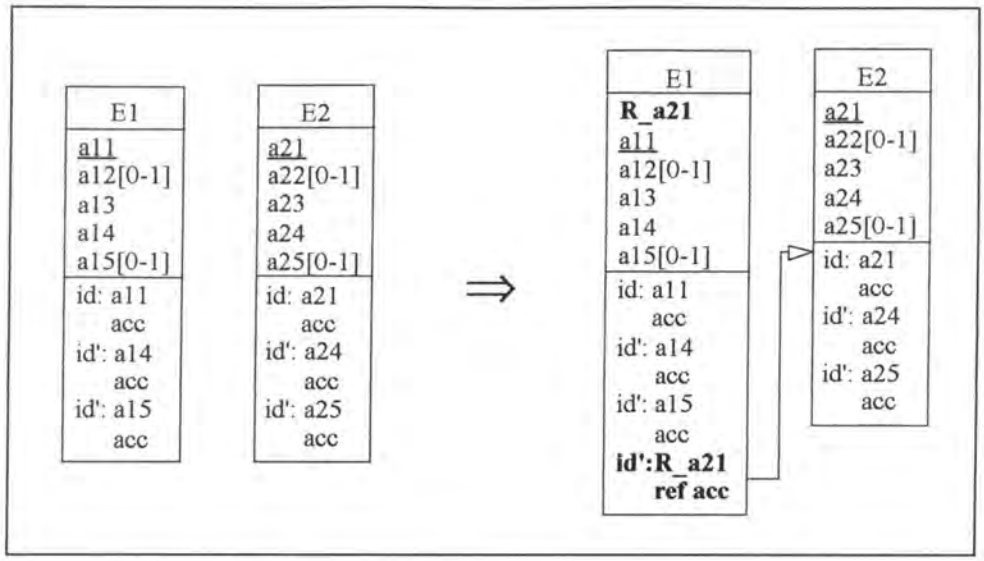


Figure A2 - 7 : Adding a 1-1/0-1 relationship-type on the logical level

2.2.1.1.2. SQL Description & Data

```
alter table E1
  add R_a21  <type> default <value>  not null  constraint E1_R_a21;

(* The user has to introduce the data into column R_a21 representing the
   relationship-type R. He must be aware that the rows of E1 which have no
   data specified for column R_a21 will be deleted. *)

delete from E1
  where R_a21 = <value>;
alter table E1
  add constraint unique (R_a21) constraint idE1_#,
  add constraint foreign key (R_a21) references E2  constraint E2_#;
```

2.2.1.1.3. Program Extracts

Note:
Sometimes select queries taken out of their program environment are not sufficient to study completely the impact on the application programs, as, for example, they do not show the changes that must be made on the variables. We therefore consider in some cases embedded queries.

Program extracts containing 'select *' must be modified. Let us consider the following program extract:

```
var a11:   <type>;
    a12:   <type>;
    a13:   <type>;
    a14:   <type>;
    a15:   <type>;
```

```
   :
exec SQL
      select *
      into :a11, :a12, :a13, :a14, :a15
      from E1
      where a14 ...
end exec
   :
```

To adapt this part of program to the changes made on table E1, we propose two potential modifications:

- We explicit the 'select *':

```
var a11:   <type>;
    a12:   <type>;
    a13:   <type>;
    a14:   <type>;
    a15:   <type>;

;
exec SQL
      select a11, a12, a13, a14, a15
      into :a11, :a12, :a13, :a14, :a15
      from E1
      where a14 ...
end exec
   :
```

- We add a variable R_a21 corresponding to the new column R_a21:

```
var a11:   <type>;
    a12:   <type>;
    a13:   <type>;
    a14:   <type>;
    a15:   <type>;
    R_a21: <type>;

   :
exec SQL
      select *
      into :a11, :a12, :a13, :a14, :a15, :R_a21
      from E1
      where a14 ...
end exec
   :
```

A similar modification is necessary if 'select *' apppears in a cursor declaration. An example illustrating this can be found in appendix 1 (see page A1-14). Note that the operation add_1-1/0-1_rel-type has a similar impact on the program extracts containing the 'insert into' command. The user interfaces may also be changed: for example, an instance of E1 may now be displayed with the instance of E2 it is linked to.

2.2.1.2. Add_0-1/0-1_rel-type

Let us suppose we want now to link entity-type E1 to entity-type E2 by a 0-1/0-1 relationship-type R.

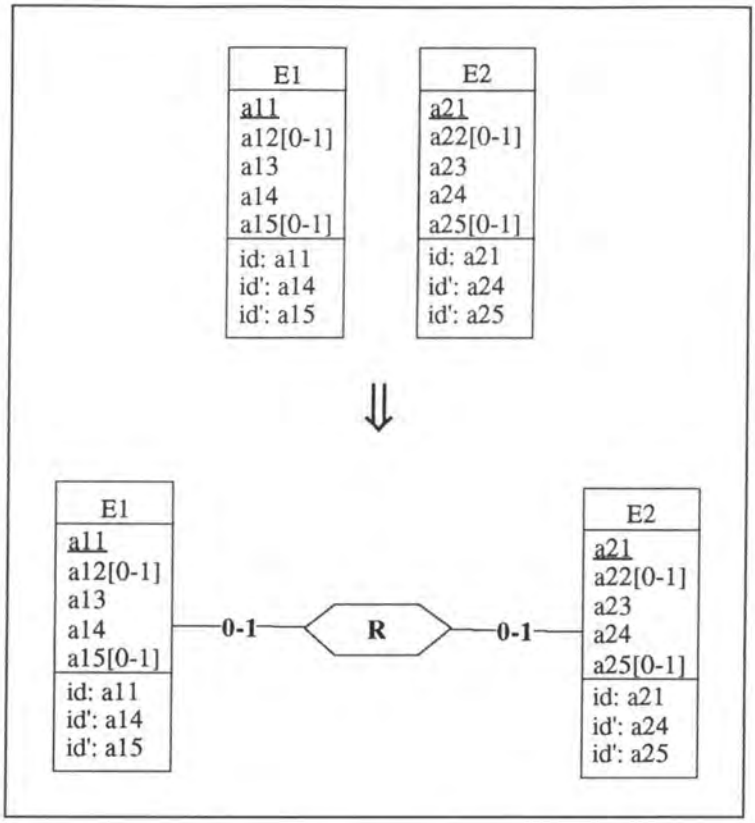


Figure A2 - 8 : Adding a 0-1/0-1 relationship-type on the conceptual level

2.2.1.2.1. Logical Schema

There are two possible representations on the logical level for the relationship-type R:

- R is implemented by a foreign key in E1
- R is implemented by a foreign key in E2

The user can choose one of the two ways of implementing R.

In the logical schema, we add either the primary key a21 of E2 to E1 or the primary key a11 of E1 to E2 as an optional foreign and candidate key.

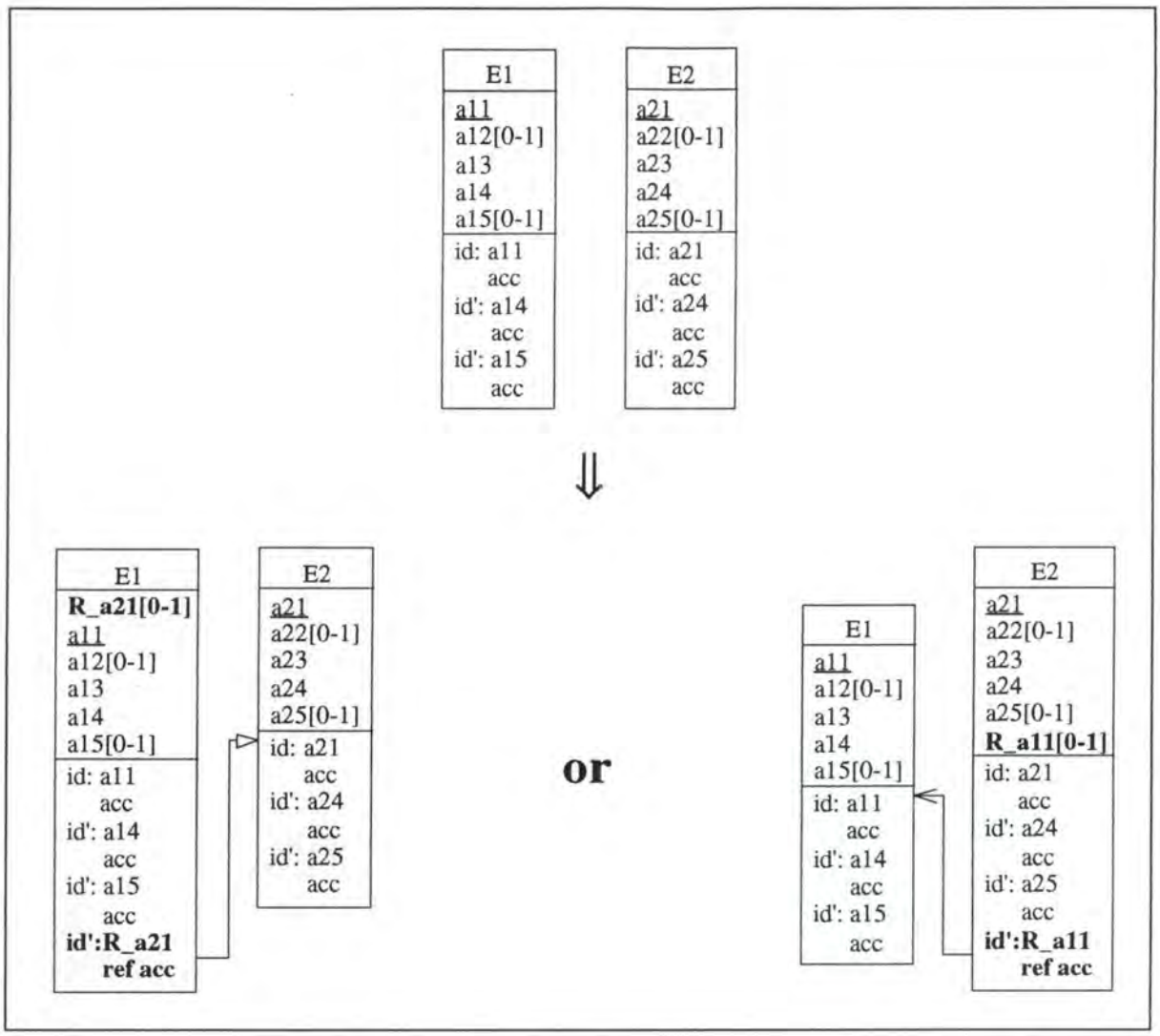


Figure A2 - 9 : Adding a 0-1/0-1 relationship-type on the logical level

2.2.1.2.2. SQL Description & Data

```
if the user wants to implement R by a foreign key in E1
then exec SQL
            alter table E1
              add R_a21    <type>,
              add constraint unique (R_a21)  constraint idE1_#,
              add constraint foreign key (R_a21) references E2
                                                    constraint E2_#;
      end exec
else (* the user wants to implement R by a foreign key in E2 *)
      exec SQL
            alter table E2
              add R_a11    <type>,
              add constraint unique (R_a11)  constraint idE2_#,
              add constraint foreign key (R_a11) references E1
                                                    constraint E1_#;
      end exec;
```

Note that either all the rows of table E1 have a null value for column R_a21 or all the rows of table E2 have a null value for column R_a11.

2.2.1.2.3. Program Extracts

We are confronted with the same problem as in the case add_1-1/0-1_rel-type (see page A2-12).

2.2.1.3. Add_1-1/0-N_rel-type

Let us suppose that we want to add a 1-1/0-N relationship-type R between entity-types E1 and E2.

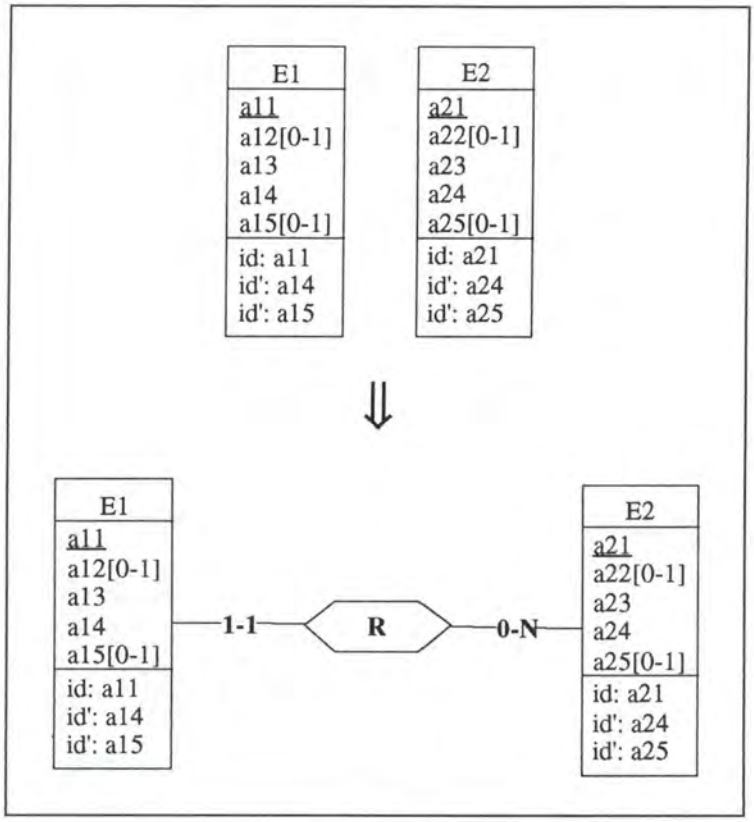


Figure A2 - 10 : Adding a 1-1/0-N relationship-type on the conceptual level

2.2.1.3.1. Logical Schema

We add the primary key a21 of E2 to E1 as a mandatory foreign key.

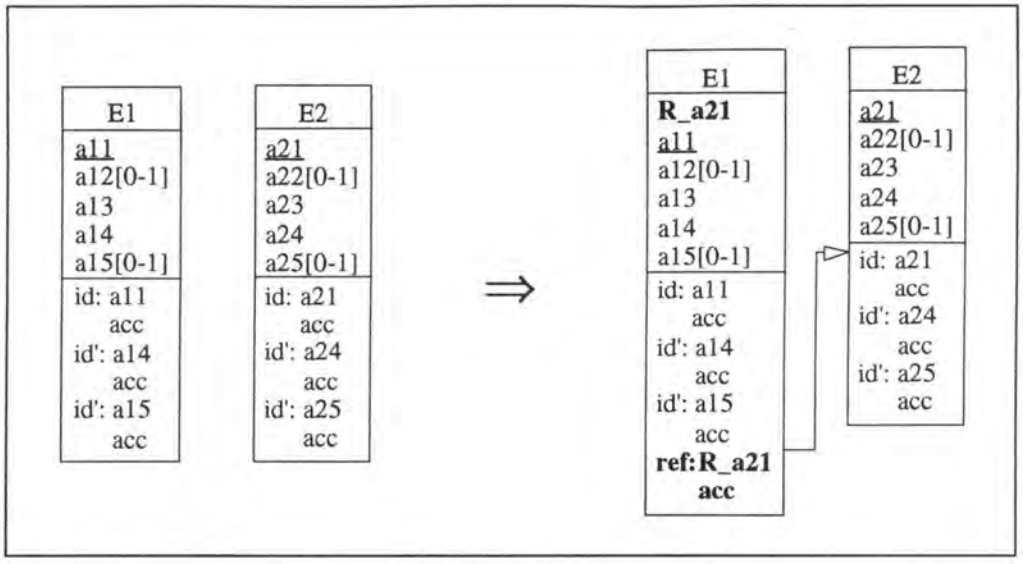


Figure A2 - 11 : Adding a 1-1/0-N relationship-type on the logical level

2.2.1.3.2. SQL Description & Data

```
alter table E1
  add R_a21  <type>  default <value>  not null constraint E1_R_a21;

(* The user has to introduce the data into column R_a21 representing the
   relationship-type R. He must be aware that the rows of E1 which have no
   data specified for column R_a21 will be deleted because of the foreign
   key constraint. *)

delete from E1
  where R_a21 = <value>;
alter table E1
  add constraint foreign key (R_a21) references E2 constraint E2_#;
```

2.2.1.3.3. Program Extracts

The modifications on the application programs are similar to those of the case add_1-1/0-1_rel-type (see page A2-12).

2.2.1.4. Add_0-1/0-N_rel-type

Let us suppose that we want this time to add a 0-1/0-N relationship-type R between the entity-types E1 and E2.

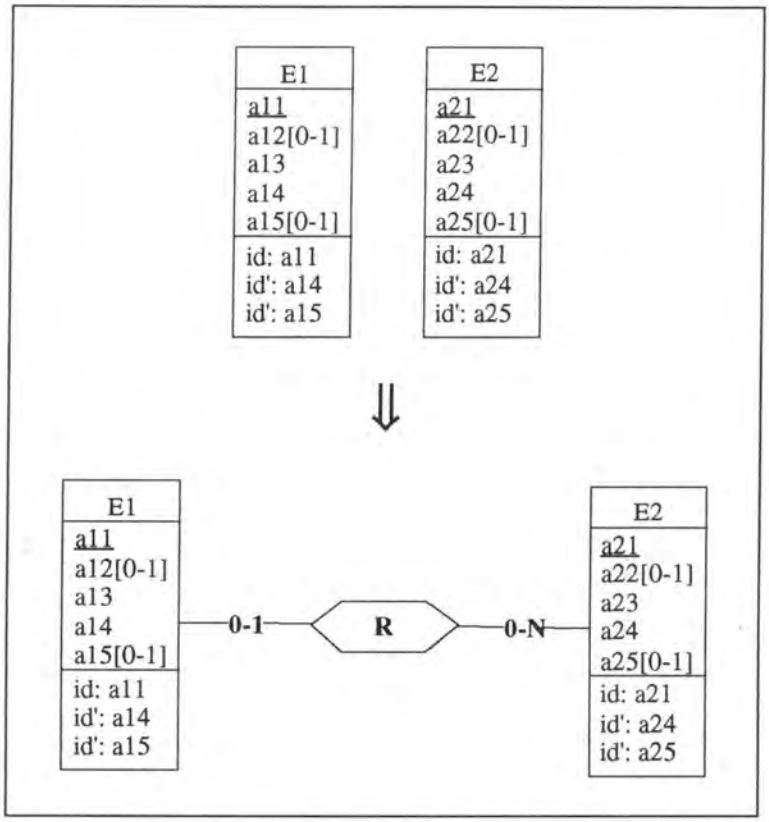


Figure A2 - 12 : Adding a 0-1/0-N relationship-type on the conceptual level

2.2.1.4.1. Logical Schema

We add the primary key a21 of E2 to E1 as an optional foreign key.

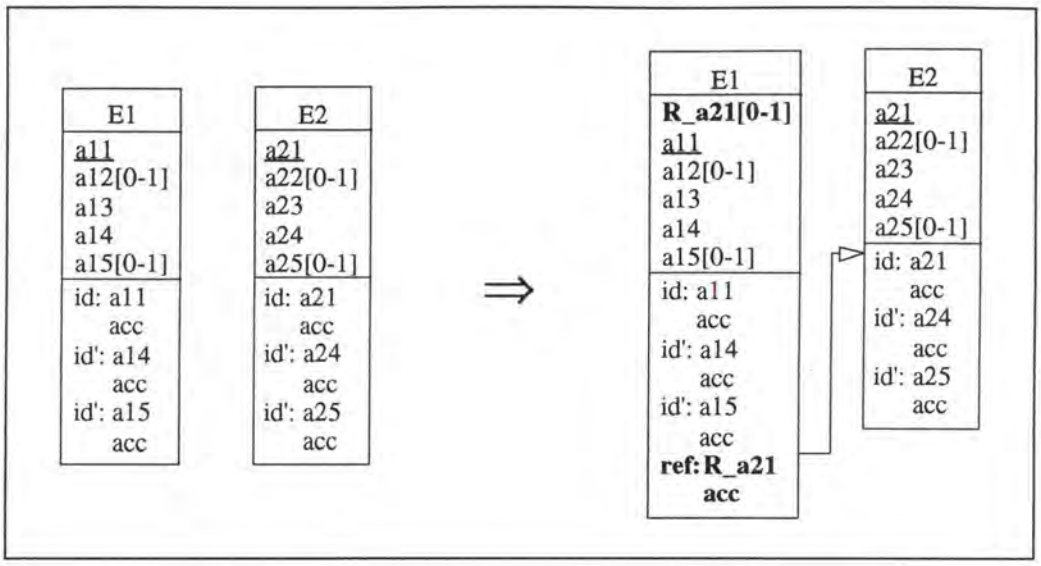


Figure A2 - 13 : Adding a 0-1/0-N relationship-type on the logical level

2.2.1.4.2. SQL Description & Data

```
alter table E1
  add R_a21  <type>,
  add constraint foreign key (R_a21) references E2  constraint E2_#;
```

2.2.1.4.3. Program Extracts

The modifications on the application programs are similar to those of the case add_1-1/0-1_rel-type (see page A2-12).

2.2.2. Modifications which Decrease the Semantics

2.2.2.1. Remove_1-1/0-1_rel-type

Let us suppose that we want to remove the 1-1/0-1 relationship-type R between the entity-types E1 and E2.

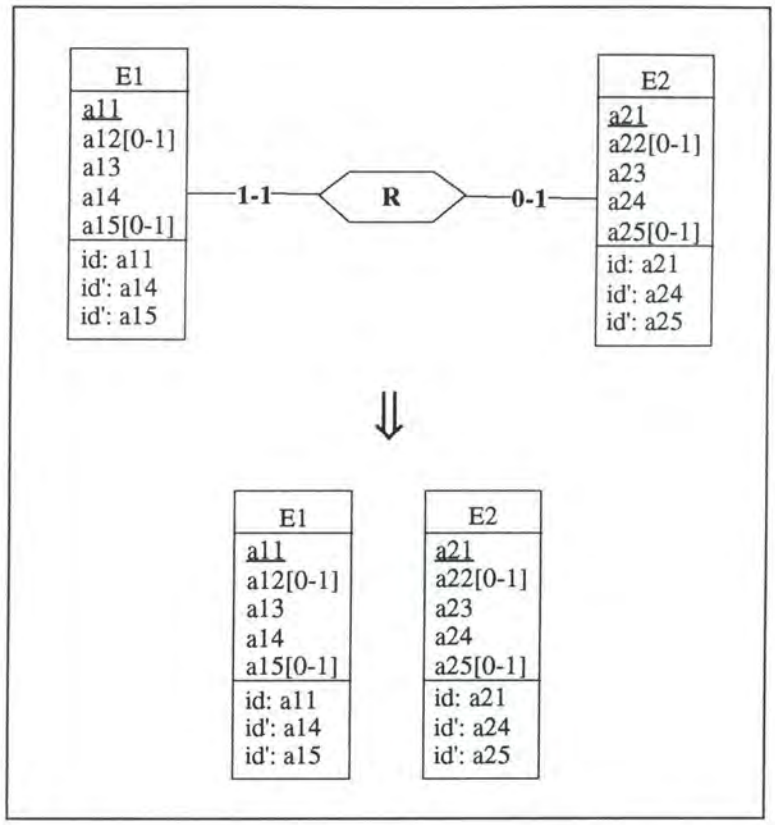


Figure A2 - 14 : Removing a 1-1/0-1 relationship-type on the conceptual level

2.2.2.1.1. Logical Schema

We remove column R_a21 from relation E1 with its candidate and foreign key features.

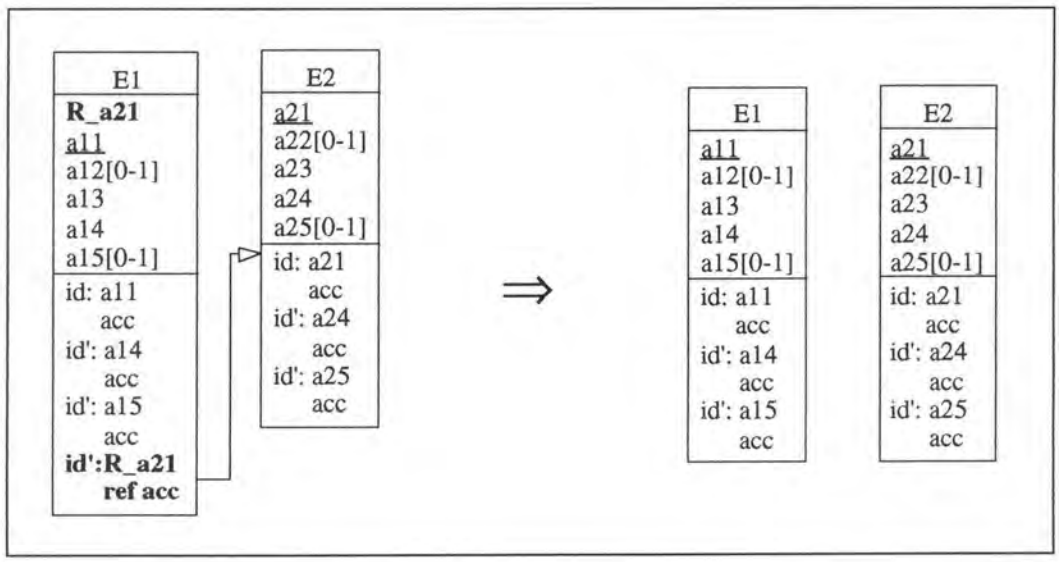


Figure A2 - 15 : Removing a 1-1/0-1 relationship-type on the logical level

2.2.2.1.2. SQL Description & Data

```
alter table E1
  drop constraint idE1_#,      (* we remove the unique key feature *)
  drop constraint E2_#,        (* we remove the foreign key feature *)
  drop constraint E1_R_a21,    (* we remove the mandatory feature of column
                                  R_a21 *)
  drop R_a21;
```

The data concerning the link between tables E1 and E2 is lost.

2.2.2.1.3. Program Extracts

All the select queries which reference R_a21 in E1 must be reviewed (for an example, see page A1-23). Application programs in which select queries referencing R_a21 in E1 appear must also be reviewed. We cannot describe a general method how to deal with these application programs as each one of them must be treated individually, depending on its context. A CASE tool offering this modification should indicate the concerned program extracts and should sometimes give hints about the way how to change them. The user has then to check whether the variables are still all needed. Finally, he must change certain user interfaces (for an example see page A1-24).

2.2.2.2. Remove_0-1/0-1_rel-type

Let us suppose that we want to remove the 0-1/0-1 relationship-type R between the entity-types E1 and E2.

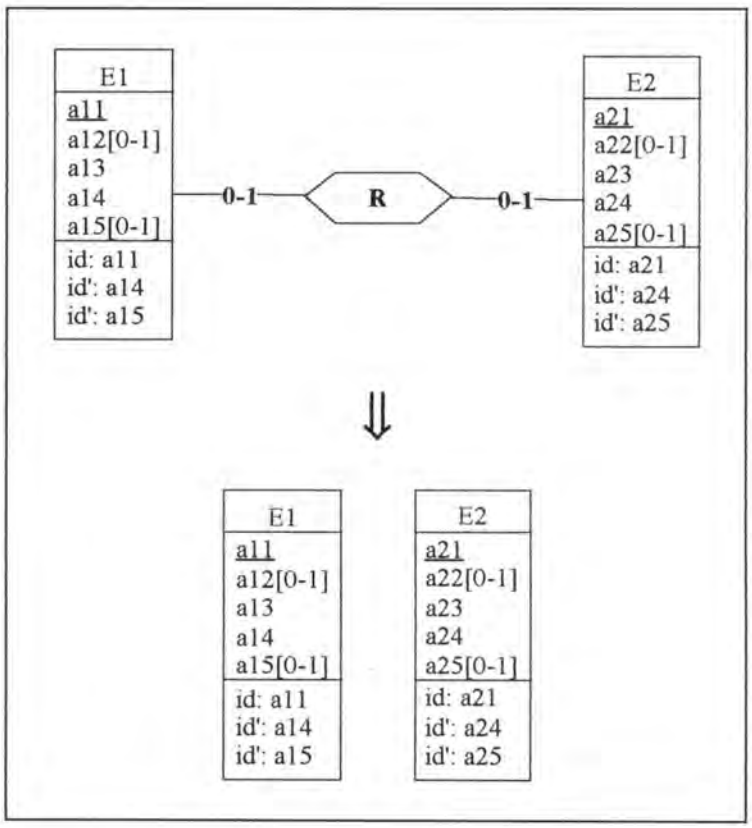


Figure A2 - 16 : Removing a 0-1/0-1 relationship-type on the conceptual level

2.2.2.2.1. Logical Schema

Depending on the way R has been implemented, we remove either column R_a21 from E1 or column R_a11 from E2 with its candidate and foreign key features.

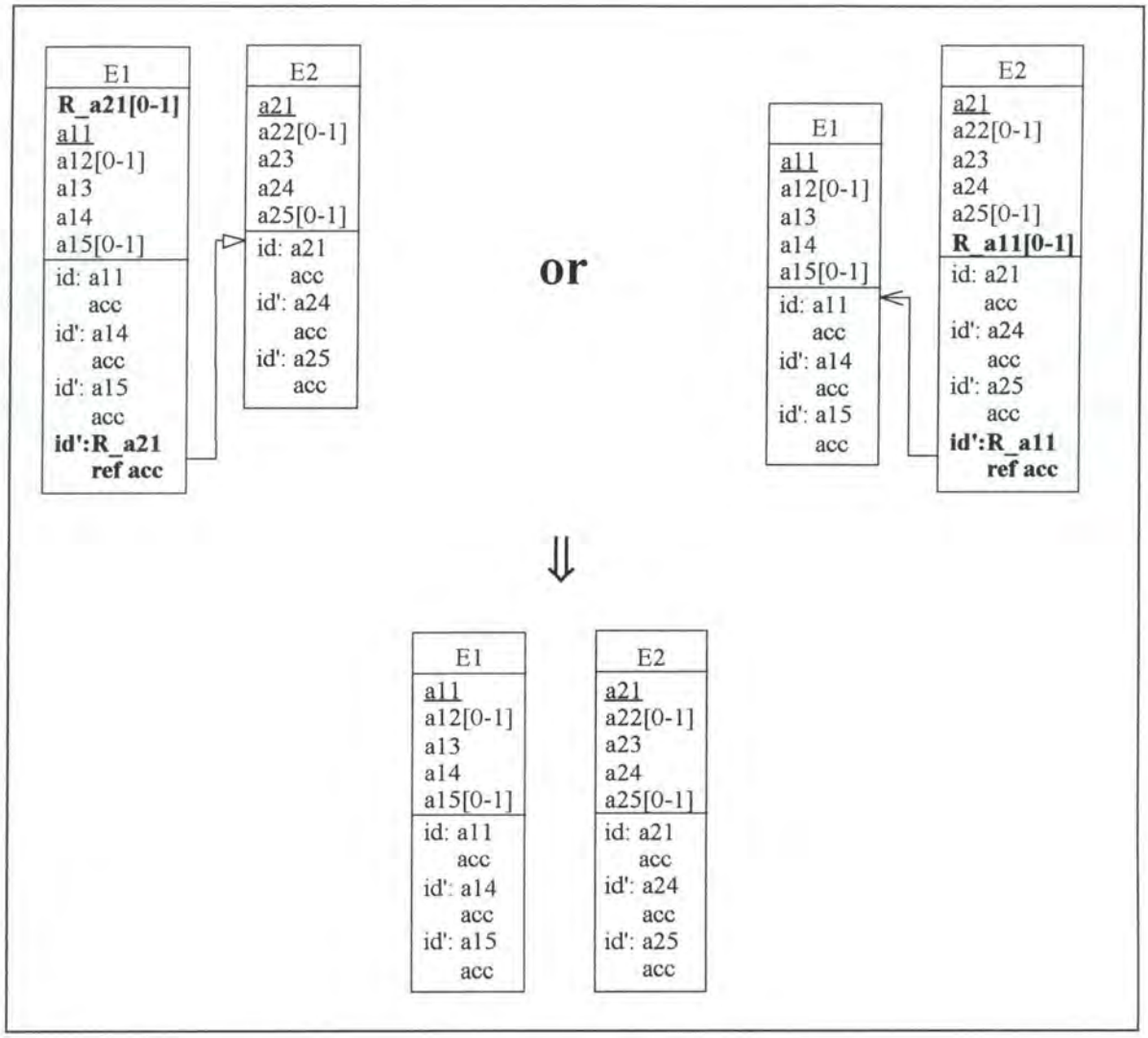


Figure A2 - 17 : Removing a 0-1/0-1 relationship-type on the logical level

2.2.2.2.2. SQL Description & Data

```
if R is implemented by a foreign key in E1
then exec SQL
           alter table E1
             drop constraint idE1_#,      (* we remove the unique key
                                             feature *)
             drop constraint E2_#,        (* we remove the foreign key
                                             feature *)
             drop R_a21;
     end exec
else   (* R is implemented by a foreign key in E2 *)
     exec SQL
           alter table E2
             drop constraint idE2_#,      (* we remove the unique key
```

```
                                        feature *)
        drop constraint E1_#,          (* we remove the foreign key
                                        feature *)

        drop R_a11;
end exec;
```

The link, representing R, between tables E1 and E2 is lost.

2.2.2.2.3. Program Extracts

Application programs in which select queires referencing R_a21 in E1 appear must be reviewed in a similar way as in the modification remove_1-1/0-1_rel-type (see page A2-21).

2.2.2.3. Remove_1-1/0-N_rel-type

Let us remove the 1-1/0-N relationship-type R between E1 and E2.

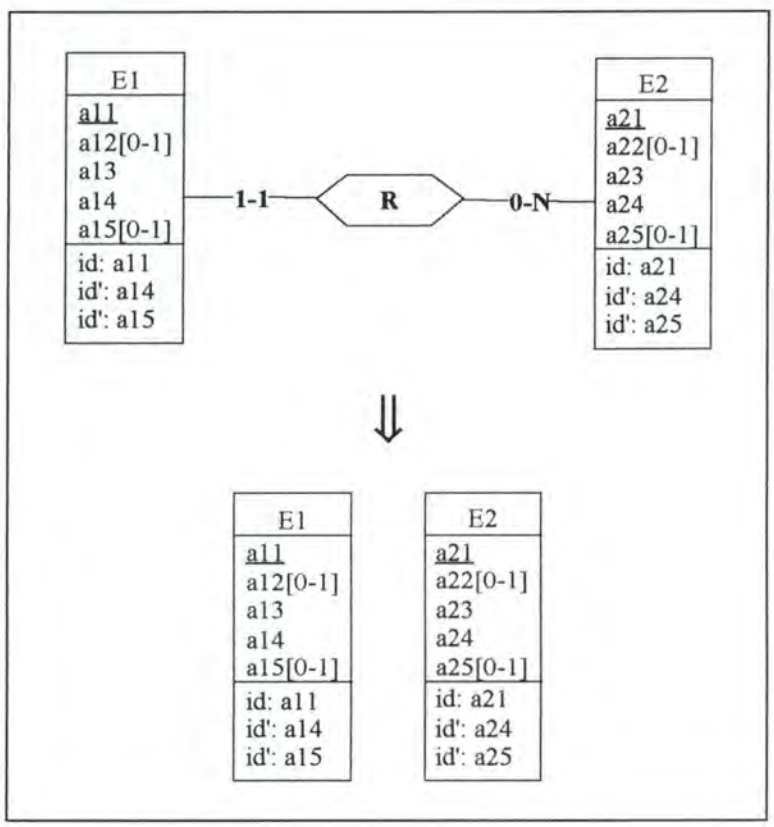


Figure A2 - 18 : Removing a 1-1/0-N relationship-type on the conceptual level

2.2.2.3.1. Logical Schema

We remove the column R_a21 in E1 with its foreign key feature.

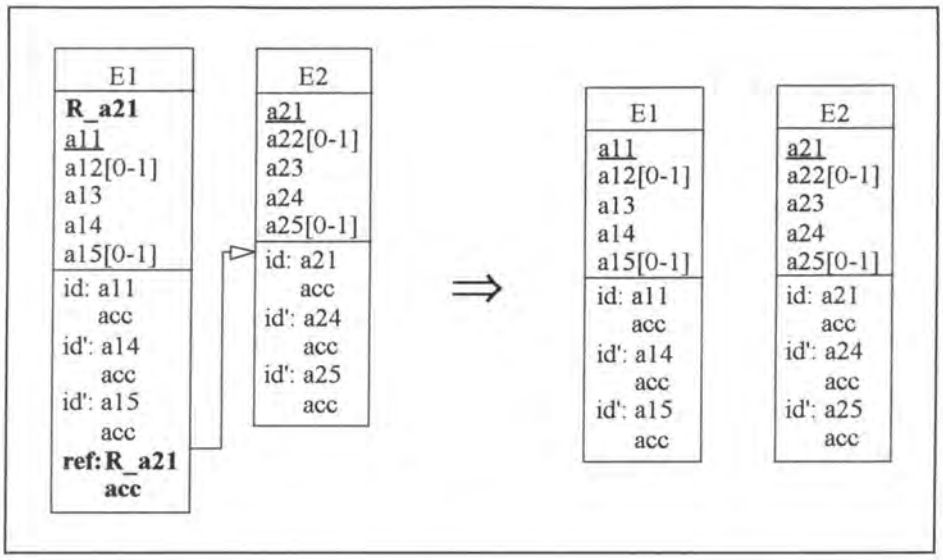


Figure A2 - 19 : Removing a 1-1/0-N relationship-type on the logical level

2.2.2.3.2. SQL Description & Data

```
alter table E1
  drop constraint E2_#,        (* we remove the foreign key feature *)
  drop constraint E1_R_a21,    (* we remove the mandatory feature from
                                  column R_a21 *)
  drop R_a21;
```

The data concerning the link, representing R, between the tables E1 and E2 is lost.

2.2.2.3.3. Program Extracts

The impacts on the application programs are similar to those of the modification remove_1-1/0-1_rel_type (see page A2-21).

2.2.2.4. Remove_0-1/0-N_rel-type

Let us remove the 0-1/0-N relationship-type R between E1 and E2.

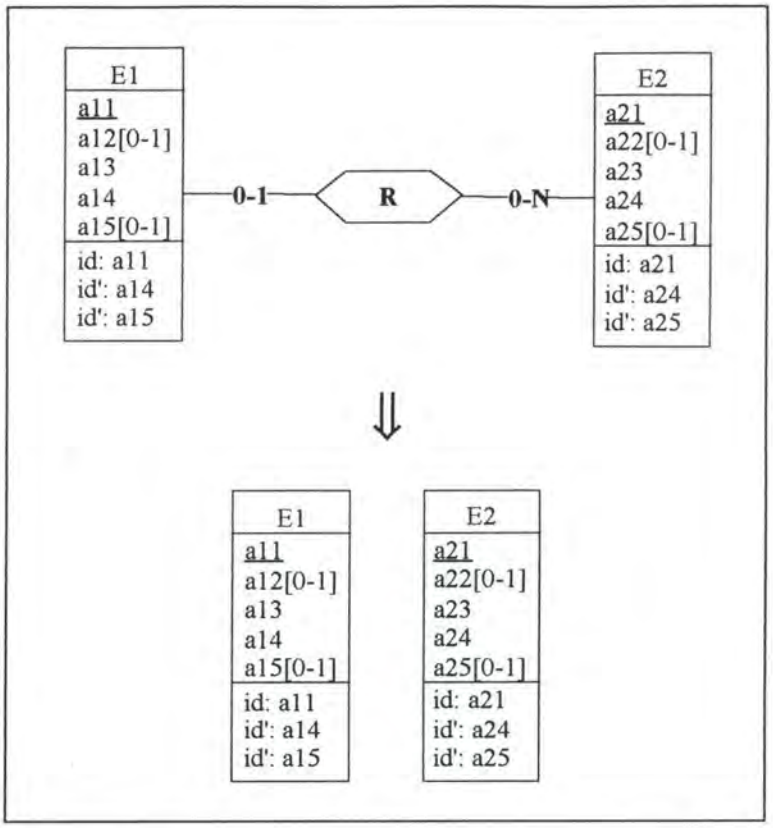


Figure A2 - 20 : Removing a 0-1/0-N relationship-type on the conceptual level

2.2.2.4.1. Logical Schema

We remove the column R_a21 in E1 with its foreign key feature.

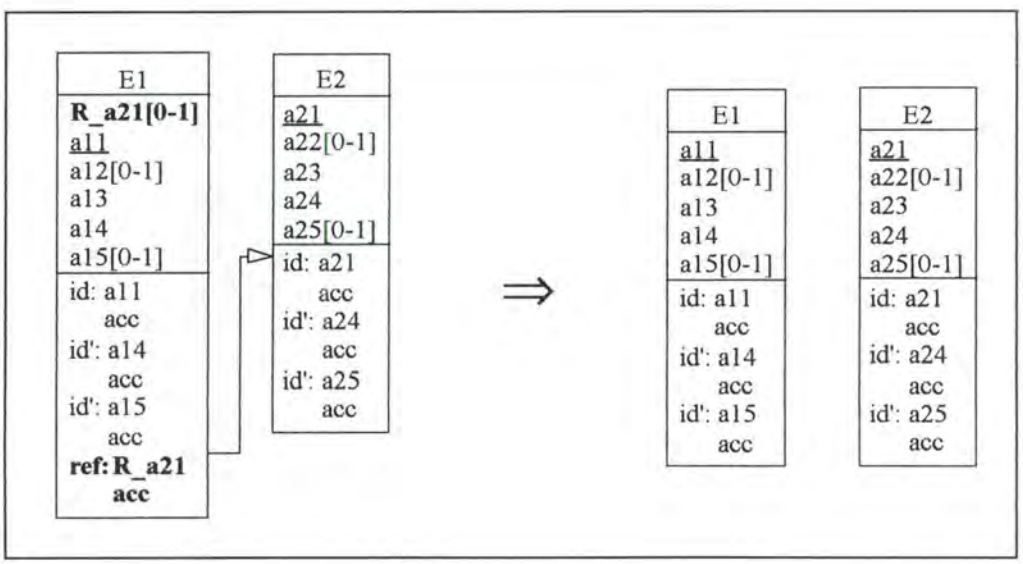


Figure A2 - 21 : Removing a 0-1/0-N relationship-type on the logical level

2.2.2.4.2. SQL Description & Data

```
alter table E1
  drop constraint E2_#, (* we remove the foreign key feature *)
  drop R_a21;
```

The data concerning the link, representing R, between the tables E1 and E2 is lost.

2.2.2.4.3. Program Extracts

The impacts on the application programs are similar to those of the modification remove_1-1/0-1_rel_type (see page A2-21).

2.2.3. Modifications which Preserve the Semantics

2.2.3.1. Rename_1-1/0-1_rel-type

Let us suppose we want to rename into R1 the relationship-type R between E1 and E2.

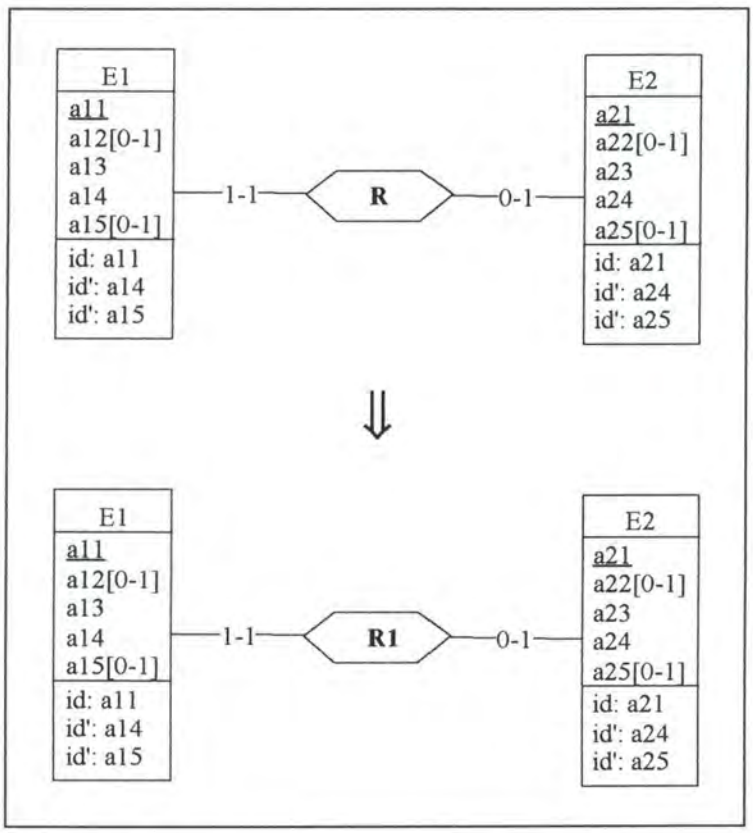


Figure A2 - 22 : Renaming a 1-1/0-1 relationship-type on the conceptual level

2.2.3.1.1. Logical Schema

On the logical level, we have to rename the foreign key column R_a21 and rename it also in the foreign and candidate key constraints.

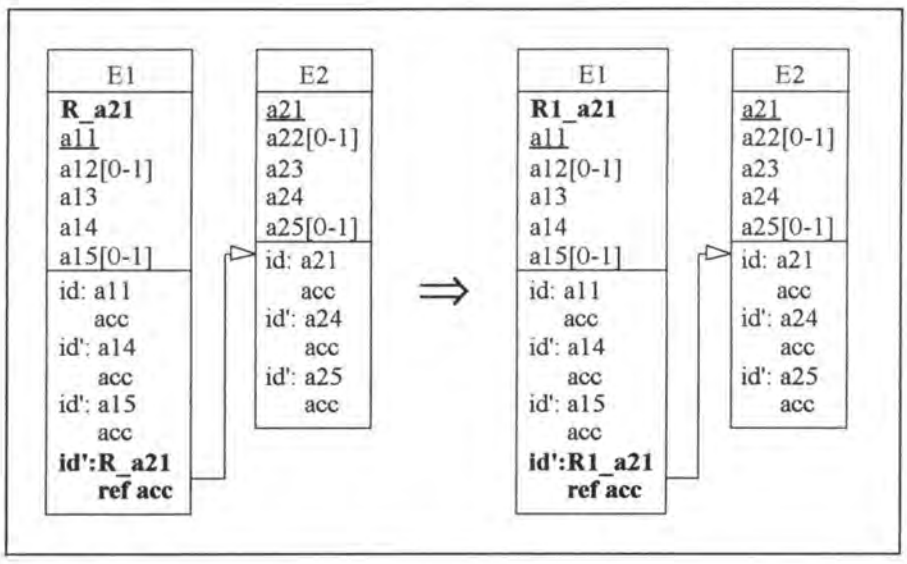


Figure A2 - 23 : Renaming a 1-1/0-1 relationship-type on the logical level

2.2.3.1.2. SQL Description & Data

```
alter table E1
  add R1_a21  <type>  default <value>  not null  constraint E1_R1_a21;
update E1
  set R1_a21 = R_a21;
alter table E1
  drop constraint E2_#,        (* we remove the old foreign key feature *)
  drop constraint idE1_#,      (* we remove the old unique key feature *)
  drop constraint E1_R_a21,    (* we remove the mandatory feature of column
                                  R_a21 *)
  add constraint unique (R1_a21) constraint idE1_#,
  add constraint foreign key (R1_a21)  references E2  constraint E2_#,
  drop R_a21;
```

This operation does not involve loss of data as the values of column R_a21 are copied into column R1_a21.

2.2.3.1.3. Program Extracts

- We have to rename R_a21 in all the select queries referencing it. For example:

```
- select R_a21
  from E1
  where ...
```

⇓

```
select R1_a21
  from E1
  where ...
```

- In addition to the select queries, we have also to review the application programs in which they appear. For instance, we must rename certain variables and/or some fields or headings in the user interfaces.

2.2.3.2. Rename_0-1/0-1_rel-type

Let us suppose that we want to rename into R1 the 0-1/0-1 relationship-type R between the entity-types E1 and E2.

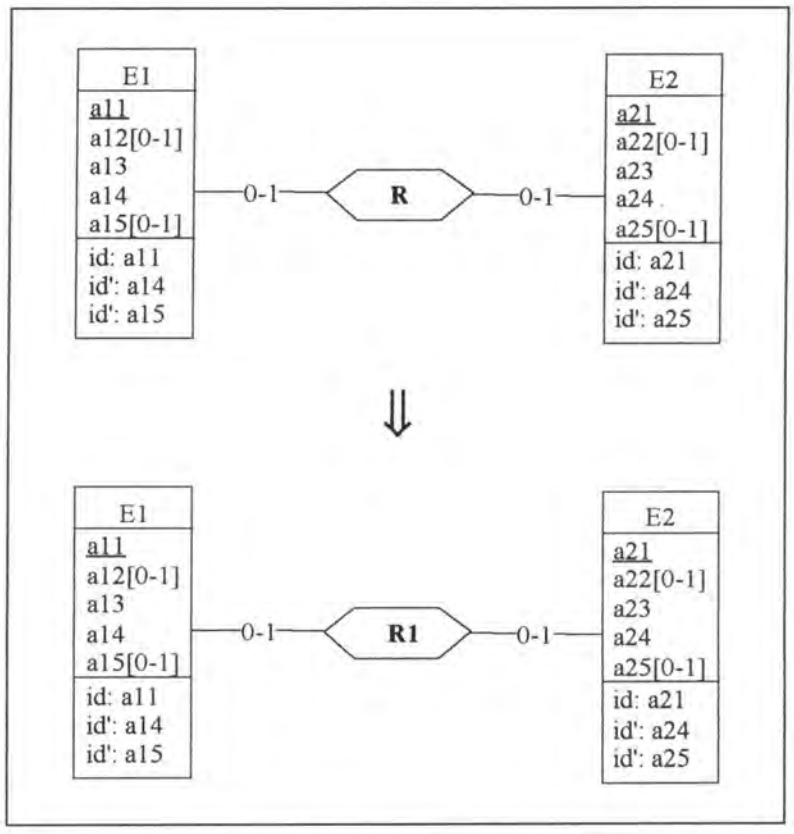


Figure A2 - 24 : Renaming a 0-1/0-1 relationship-type on the conceptual level

2.2.3.2.1. Logical Schema

Depending on the way R has been implemented, we rename either column R_a21 in E1 or R_a11 in E2 and in their respective candidate and foreign key constraints.

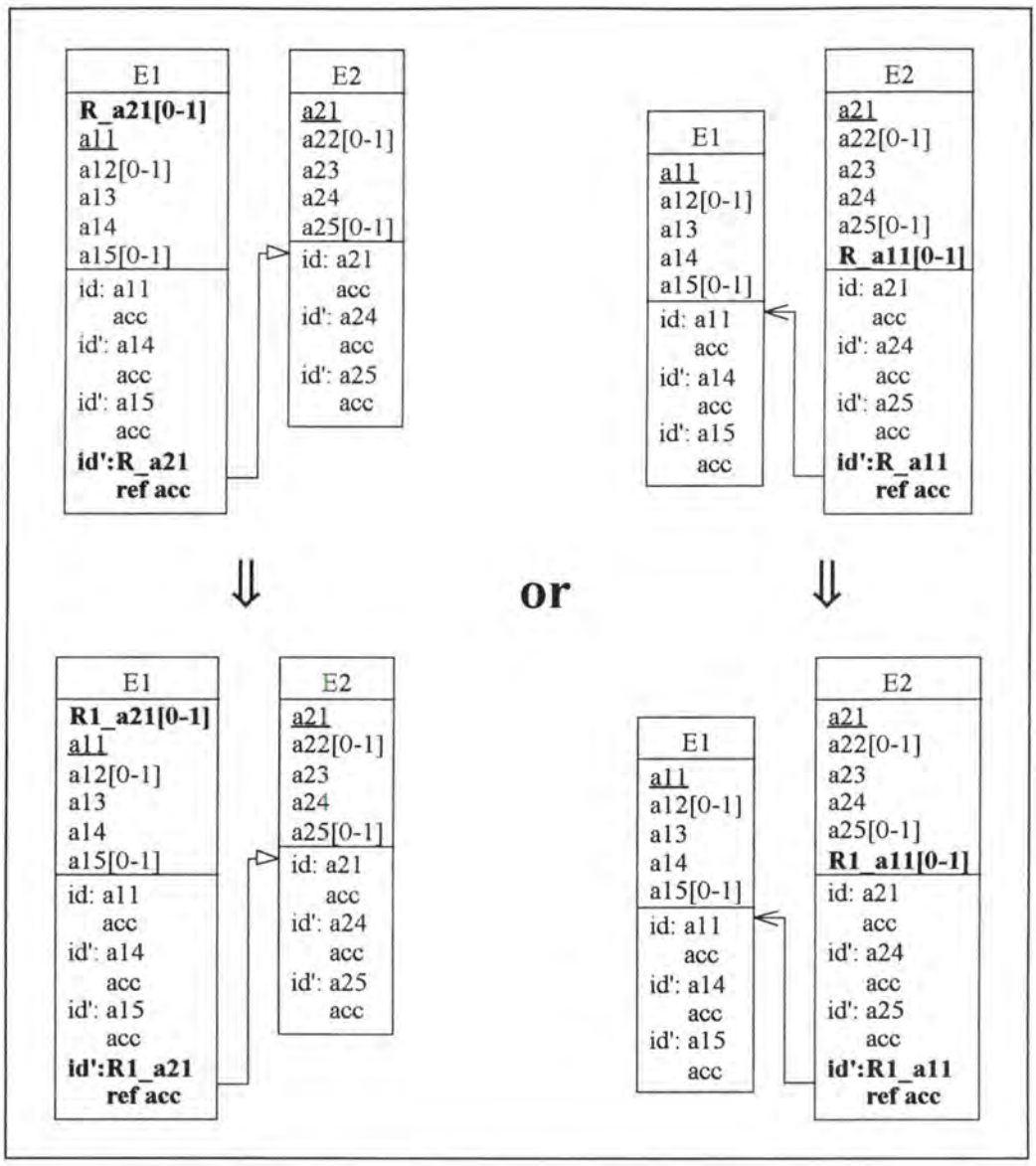


Figure A2 - 25 : Renaming a 0-1/0-1 relationship-type on the logical level

2.2.3.2.2. SQL Description & Data

```
if R is implemented by a foreign key in E1
then exec SQL
          alter table E1
            add R1_a21    <type>;
          update E1
            set R1_a21 = R_a21;
          alter table E1
            drop constraint E2_#, (* we remove the old foreign key
                                     feature *)
            drop constraint idE1_#,      (* we remove the old unique key
                                            feature *)
            add constraint unique (R1_a21) constraint idE1_#,
            add constraint foreign key (R1_a21) references E2
                                                  constraint E2_#,
            drop R_a21;
     end exec
```

```
else (* R is implemented by a foreign key in E2 *)
      exec SQL
            alter table E2
              add R1_a11    <type>;
            update E2
              set R1_a11 = R_a11;
            alter table E2
              drop constraint E1_#, (* we remove the old foreign key
                                       feature *)
              drop constraint idE2_#,     (* we remove the old unique key
                                             feature *)
              add constraint unique (R1_a11) constraint idE2_#,
              add constraint foreign key (R1_a11) references E1
                                                    constraint E1_#,
              drop R_a11;
      end exec;
```

This operation does not involve loss of data, as the values of column R_a11 are copied into column R1_a11.

2.2.3.2.3. Program Extracts

The impacts on the application programs are similar to those of the modification rename_1-1/0-1_rel-type (see page A2-27).

2.2.3.3. Rename_1-1/0-N_rel-type

Let us rename the 1-1/0-N relationship-type R between E1 and E2 into R1.

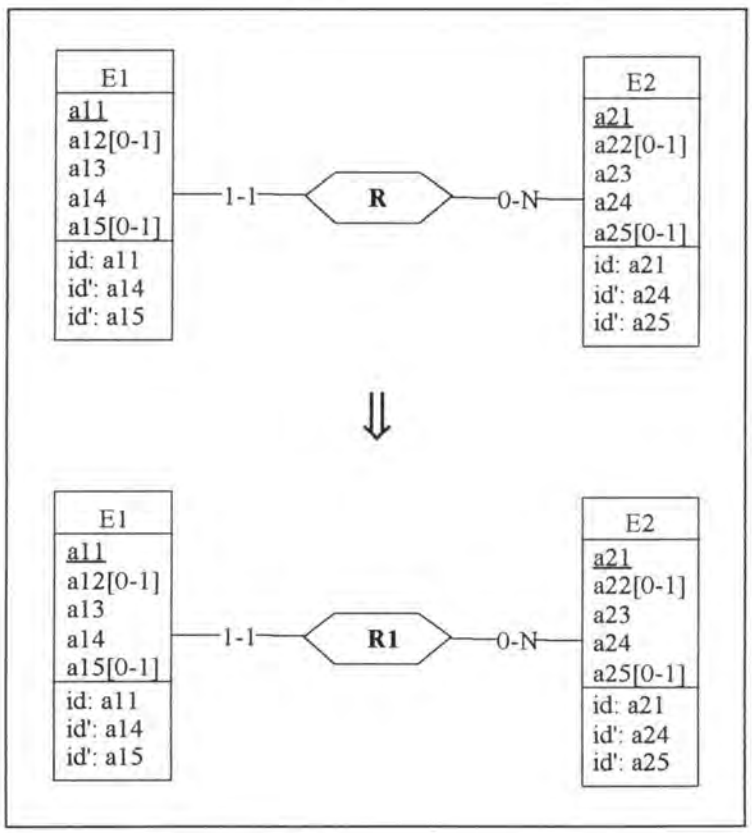


Figure A2 - 26 : Renaming a 1-1/0-N relationship-type on the conceptual level

2.2.3.3.1. Logical Schema

In the logical schema, the foreign key column R_a21 must be renamed in E1 and in its foreign key constraint.

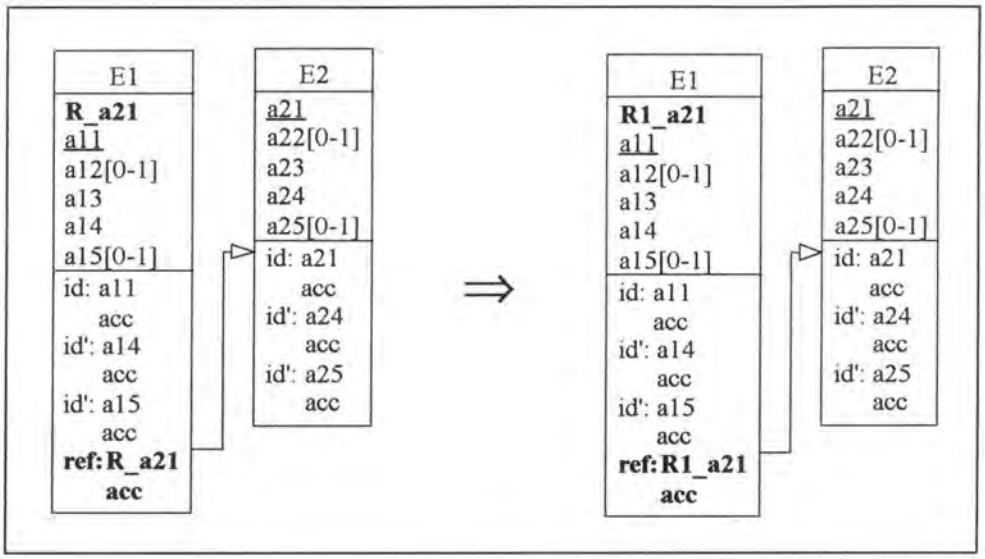


Figure A2 - 27 : Renaming a 1-1/0-N relationship-type on the logical level

2.2.3.3.2. SQL Description & Data

```
alter table E1
  add R1_a21    <type>  default <value>  not null  constraint E1_R1_a21;
update E1
  set R1_a21 = R_a21;
alter table E1
  drop constraint E2_#, (* we remove the old foreign key feature *)
  drop constraint E1_R_a21,    (* we remove the mandatory feature from
                                  column R_a21 *)
  add constraint foreign key (R1_a21)  references E2  constraint E2_#,
  drop R_a21;
```

This operation does not involve loss of data as the values of column R_a21 are copied into column R1_a21.

2.2.3.3.3. Program Extracts

The impacts on the program extracts are similar to those of the modification rename_1-1/0-1_rel-type (see page A2-27).

2.2.3.4. Rename_0-1/0-N_rel-type

Let us rename the 0-1/0-N relationship-type R between E1 and E2 into R1.

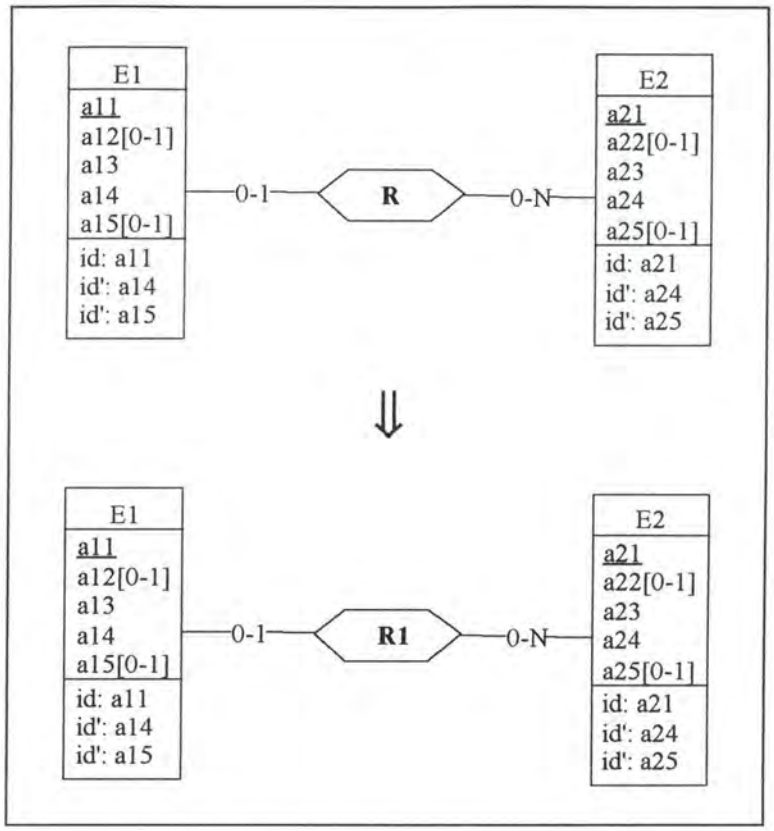


Figure A2 - 28 : Renaming a 0-1/0-N relationship-type on the conceptual level

2.2.3.4.1. Logical Schema

In the logical schema, the foreign key column R_a21 must be renamed in E1 and in its foreign key constraint.

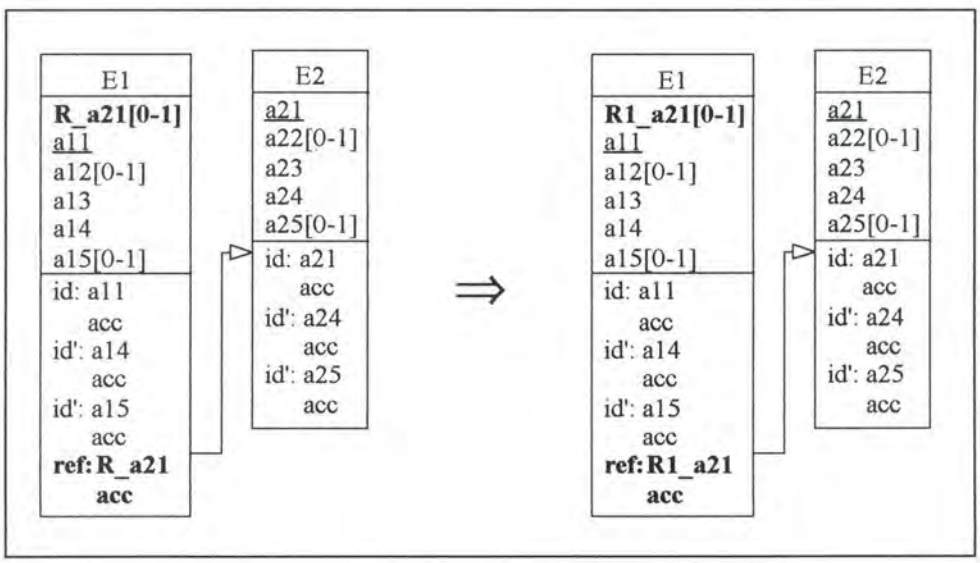


Figure A2 - 29 : Renaming a 0-1/0-N relationship-type on the logical level

2.2.3.4.2. SQL Description & Data

```
alter table E1
  add R1_a21    <type>;
update E1
  set R1_a21 = R_a21;
alter table E1
  drop constraint E2_#, (* we remove the old foreign key feature *)
  add constraint foreign key (R1_a21)  references E2  constraint E2_#,
  drop R_a21;
```

This operation does not involve loss of data as the values of column R_a21 are copied into column R1_a21.

2.2.3.4.3. Program Extracts

The impacts on the program extracts are similar to those of the modification rename_1-1/0-1_rel-type (see page A2-27).

2.3. MODIFICATIONS OF THE ROLES

2.3.1. Modifications which Augment the Semantics

2.3.1.1. Augment_max_card

Precondition:
Given the restrictions of the relationship-types in the Kernel (see page 3-2), the only augmentations of the maximum cardinality of a role that we accept so far are:

- 1-1/0-1 → 1-1/0-N
- 0-1/0-1 → 0-1/0-N

We want to augment to N the maximum cardinality of the 0-1 role of relationship-type R.

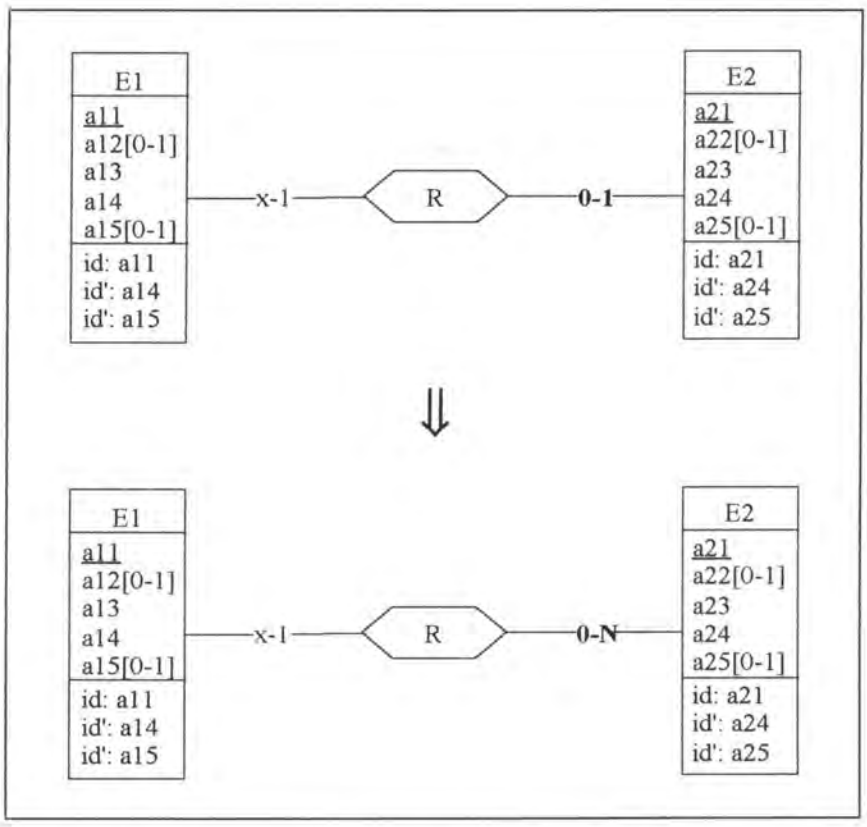


Figure A2 - 30 : Augmenting the maximum cardinality of a role to N on the conceptual level

2.3.1.1.1. Logical Schema

We have either to remove the candidate key from R_a21 in E1 or to replace the foreign key R_a11 in E2 by a (non unique) foreign key R_a21 in E1.

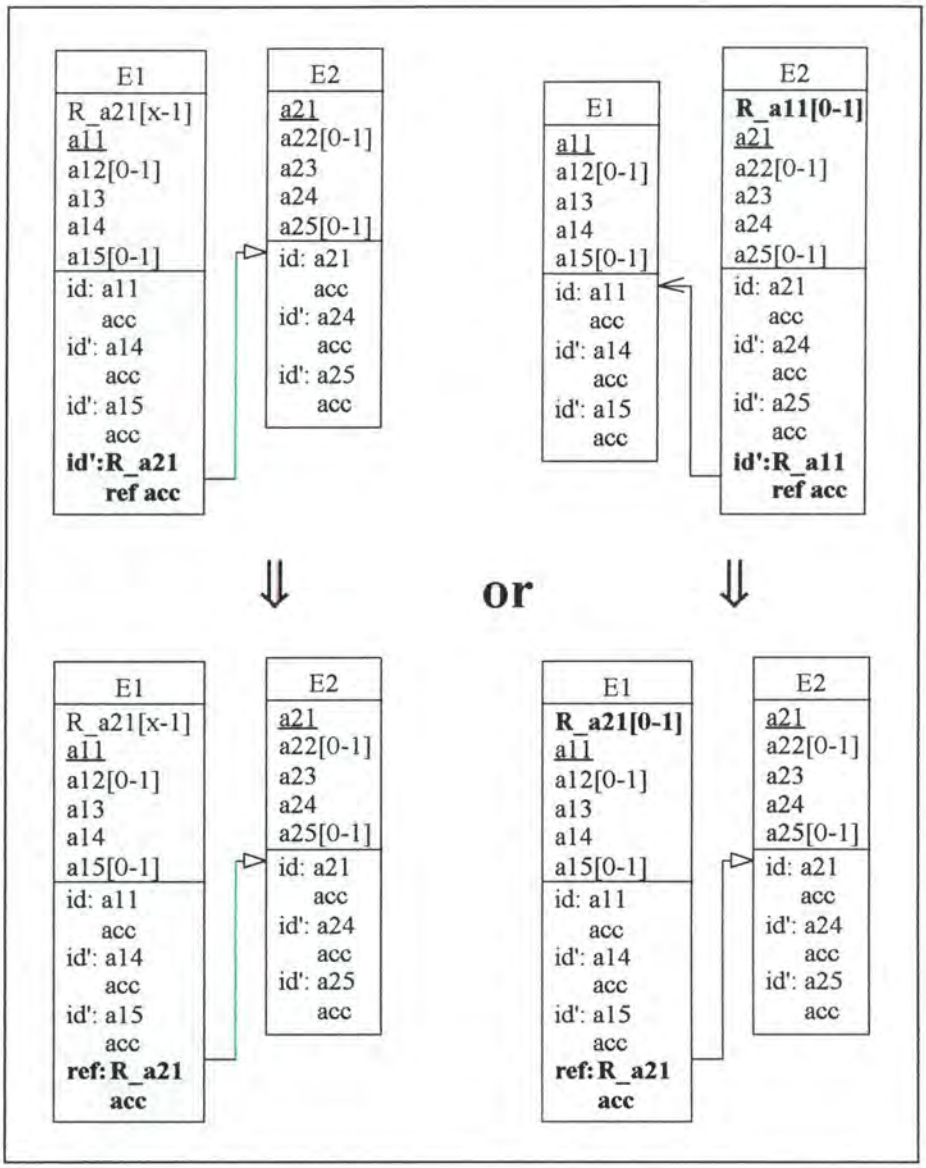


Figure A2 - 31 : Augmenting the maximum cardinality of a role to N on the logical level

2.3.1.1.2. SQL Description & Data

```
var a21: <type>;
    a11: <type>;

if the foreign key representing R is in E1
then exec SQL
           alter table E1
             drop constraint idE1_#;
     end exec
```

```
else (* the foreign key representing R is in E2 *)
      begin
        exec SQL
            (* we create the new foreign key column *)
            alter table E1
              add R_a21  <type>;
            (* we copy the data representing the relationship-type R from
               table E2 into table E1 *)
            declare c cursor for
              select a21, R_a11
                from E2
                where R_a11 is not null;
            open c;
            fetch c into :a21, :a11;
        end exec;
        while SQLCODE = 0      (* the last item has not yet been treated *)
        do begin
              exec SQL
                  update E1
                    set R_a21 = :a21
                    where a11 = :a11;
                  fetch c into :a21, :a11;
              end exec;
           end;
        exec SQL
            (* we add and remove the necessary constraints *)
            alter table E1
              add constraint foreign key (R_a21) references E2
                                                    constraint E2_#;
            alter table E2
              drop constraint idE2_#,
              drop constraint E1_#,
              drop R_a11;
            close c;
        end exec;
    end;
```

Note that no data is lost as either no changes are applied on the data or the data is only 'copied' from relation E2 into relation E1.

2.3.1.1.3. Program Extracts

Before considering the select queries, let us note that the user has to replace certain variables by arrays, that he has to review certain user interfaces and that he has also to update the documentation. In order to study the impact of the modification on the select queries, we must distinguish whether the foreign key representing R was in E1 or E2.

2.3.1.1.3.1. The foreign key representing R was in E1

As the foreign key R_a21 is not identifier of E1 anymore, several rows can now have the same value for column R_a21. We thus have to define a cursor for 'select ...into ...' queries referencing 'R_a21 = ' in their 'where' clause.

```
var ...

:
exec SQL
     select ...
       into ...
       from E1
       where R_a21 = ...
end exec;
```

```
if SQLCODE = 0          (* if such a row has been found *)
then ...
:
```

```
var ...

   :
exec SQL
      declare c cursor for
        select ...
          from E1
          where R_a21 = ...;
      open c;
      fetch c into ...;
end exec;
while SQLCODE = 0       (* the last item has not yet been treated *)
  do begin
        :
        exec SQL
            fetch c into ...;
        end exec
     end;
exec SQL
      close c
end exec;
   :
```

2.3.1.1.3.2. The foreign key representing R was in E2

- A similar problem concerning the 'select ...into ...' queries occurs in this case. The select query must here however also be modified.

```
var a11: <type>;

  :
exec SQL
      select R_a11
        into :a11
        from E2
        where a24 = ...;
end exec;
if SQLCODE = 0
then ...
  :
```

```
var a11: <type>;

  :
exec SQL
      declare c cursor for
        select a11
          from E1
          where R_a21 in (select a21
                                  from E2
                                  where a24 = ...);
      open c;
      fetch c into :a11;
end exec;
while SQLCODE = 0       (* the last item has not yet been treated *)
do begin
       :
```

```
      exec SQL
           fetch c into :a11;
      end exec;
   end;
exec SQL
     close c
end exec;
  :
```

- As R is now represented by a foreign key in E1, the select queries referencing R_a11 must be reviewed.

```
- select ...
     from E1
     where a11 in ( select R_a11
                    from E2
                    where ... )
```

```
  select ...
     from E1
     where R_a21 in ( select a21
                      from E2
                      where ... )

- select ...
    from E2
    where R_a11 ...
```

```
  select ...
    from E2
    where a21 in ( select R_a21
                   from E1
                   where a11 ... )
```

2.3.1.2. Decrease_min_card

Precondition:
Given the restrictions of the relationship-types in the Kernel (see page 3-2), the only decreases of the minimum cardinality of a role that we accept so far are:

- 1-1/0-1 → 0-1/0-1
- 1-1/0-N → 0-1/0-N

We want to decrease to 0 the minimum cardinality of the 1-1 role of relationship-type R.

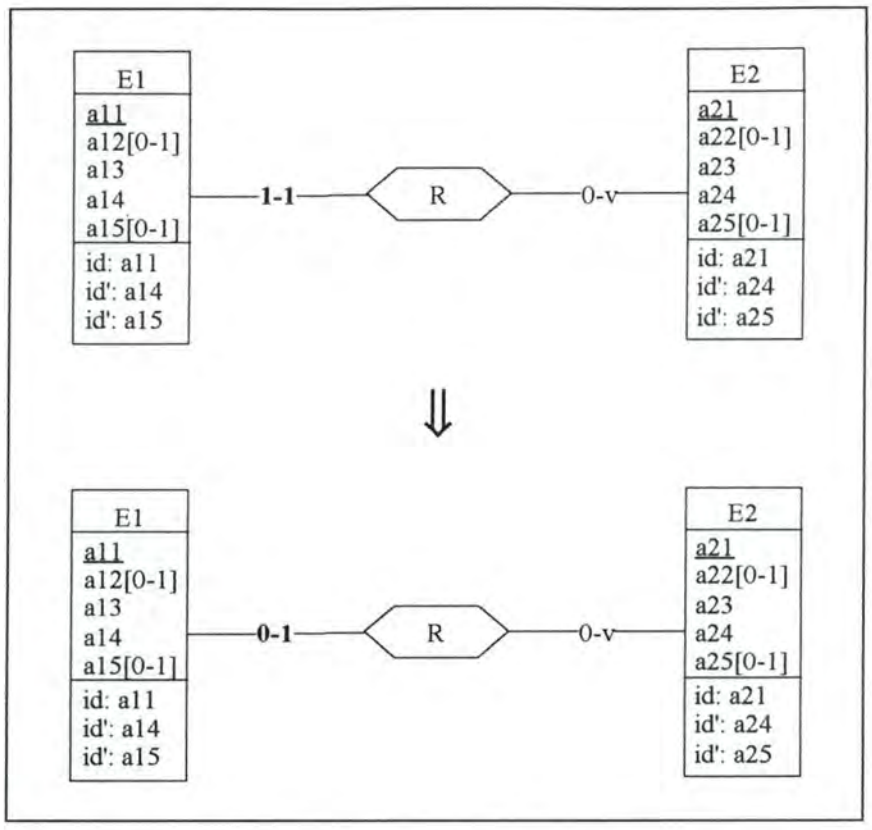


Figure A2 - 32 : Decreasing the minimum cardinality of a role to 0 on the conceptual level

2.3.1.2.1. Logical Schema

We have to make the foreign key R_a21 in E1 optional.

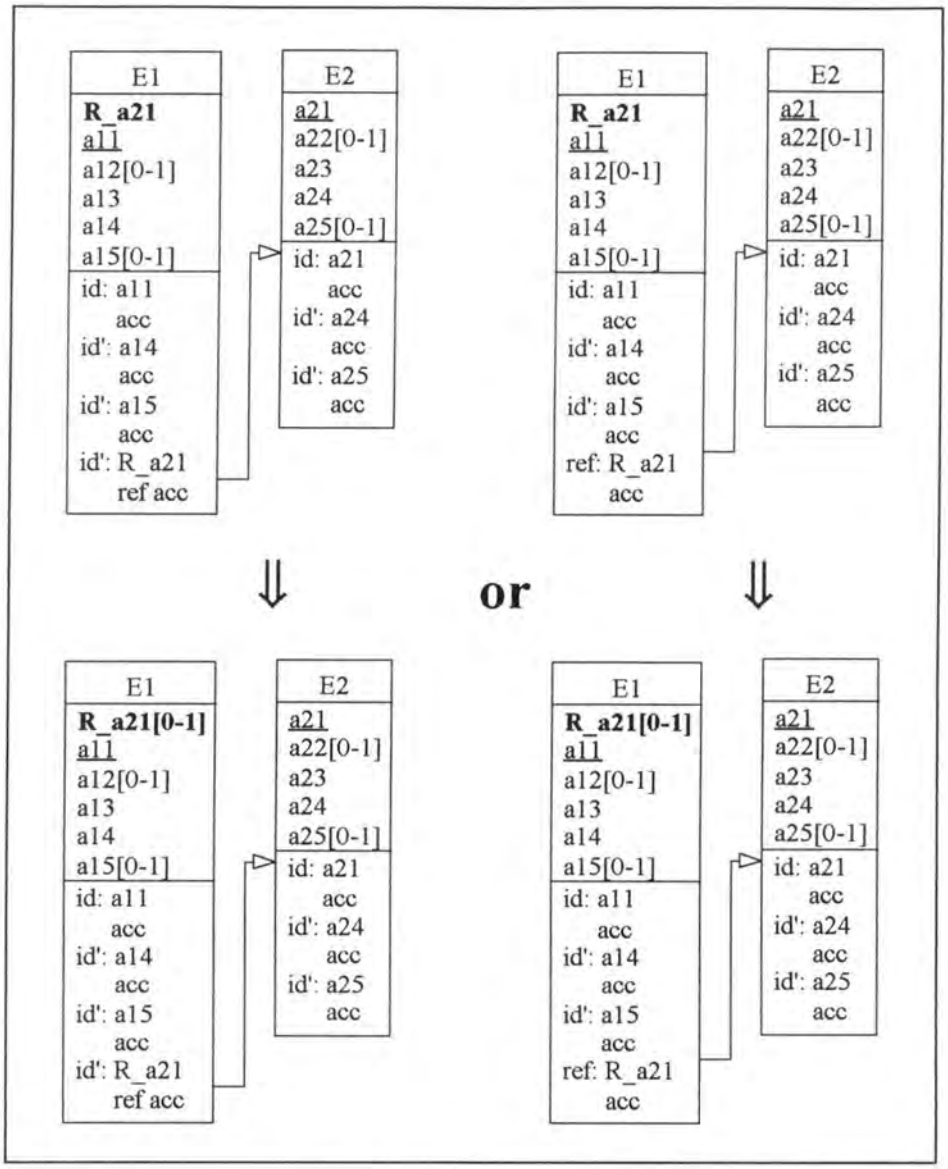


Figure A2 - 33 : Decreasing the minimum cardinality of a role to 0 on the logical level

2.3.1.2.2. SQL Description & Data

```
alter table E1
  drop constraint E1_R_a21;
```

Note that no data is lost as the foreign key is only made optional.

2.3.1.2.3. Program Extracts

The result of select queries must now be tested whether they have null values or not.

```
var a21: <type>;
```

```
  :
exec SQL
      select R_a21, ...
        into :a21, ...
        from E1
        where a11 = ...;
end exec;
if SQLCODE = 0     (* if such a row has been found *)
then ...
```

```
var a21: <type>;
    null_indicator: INTEGER;

  :
exec SQL
      select R_a21, ...
        into :a21:null_indicator, ...
        from E1
        where a11 = ...;
end exec;
if SQLCODE = 0     (* if such a row has been found *)
then if null_indicator = 0
     then ...
```

As we can see in the previous program extracts, tests (if - then clauses) checking the null value of column LIVE_ncust must sometimes be introduced. A similar change can be performed for 'select R_a21' queries which occur in cursor declarations.

2.3.2. Modifications which Decrease the Semantics

2.3.2.1. Decrease_max_card

Precondition:
Given the restrictions of the relationship-types in the Kernel (see page 3-2), the only decreases of the maximum cardinality of a role that we accept so far are:

- 1-1/0-N → 1-1/0-1
- 0-1/0-N → 0-1/0-1

We want to decrease to 1 the maximum cardinality of the 0-N role of the relationship-type R.

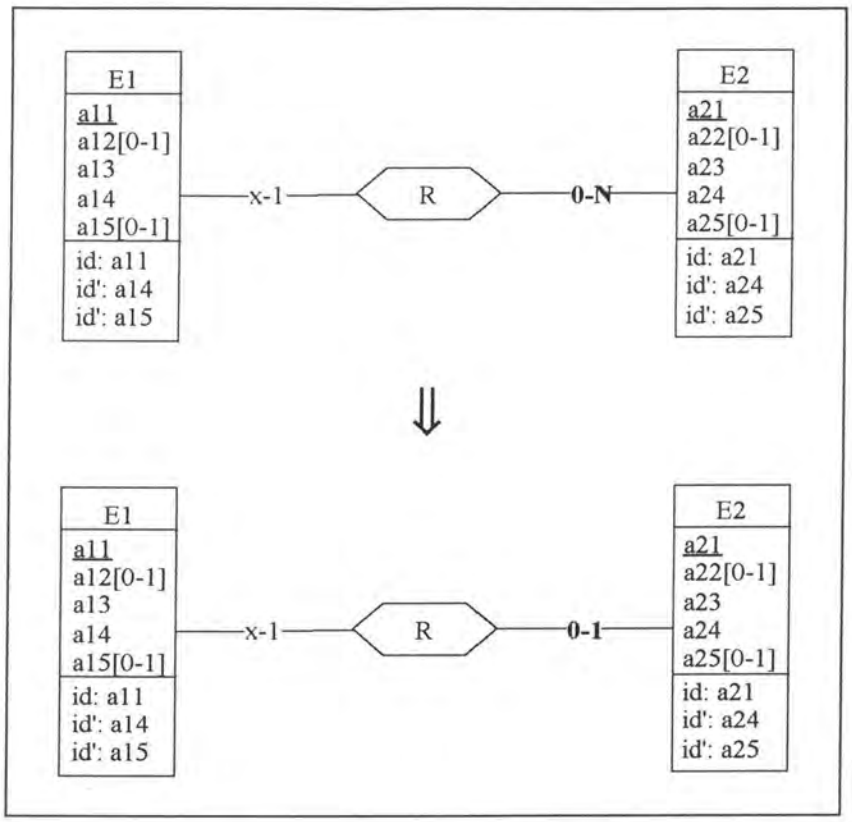


Figure A2 - 34 : Decreasing the maximum cardinality of a role on the conceptual level

2.3.2.1.1. Logical Schema

On the logical level, we have to add the candidate key feature to column R_a21 in relation E1.

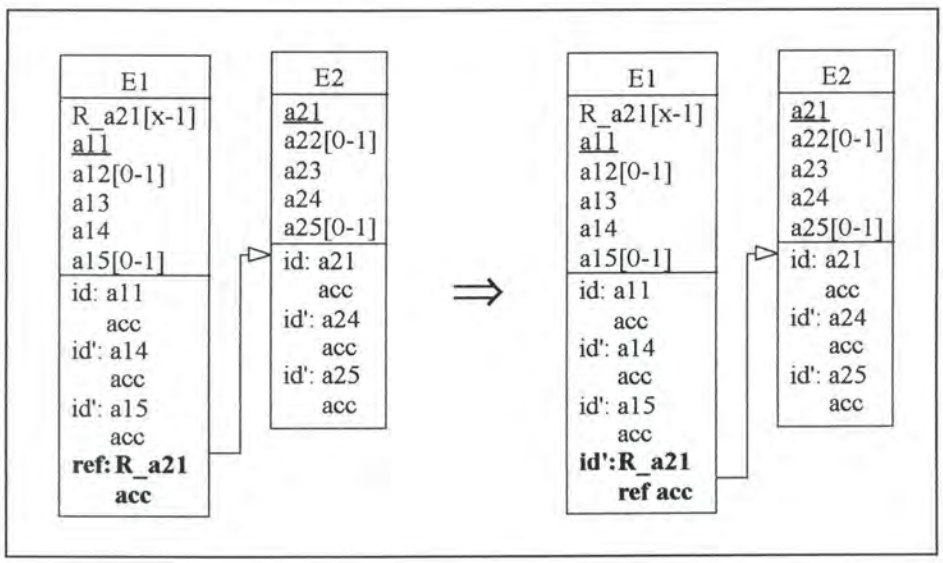


Figure A2 - 35 : Decreasing the maximum cardinality of a role on the logical level

2.3.2.1.2. SQL Description & Data

```
var a21_1: <type>;
    a21_2: <type>;
    a11:   <type>;
    count: INTEGER;

procedure Delete_on_cascade(r, E);

(* Before deleting a row r in table E, we must delete the rows r1
   'referencing r' or set to null in table E1 the foreign key column of the
   rows r1 'referencing r'. If the rows r1 are deleted, the problem must be
   treated recursively. *)

begin
  for each table E1
  do for each foreign key referencing table E
     do for each of the rows r1 having as foreign column value the value of
           the primary key column of row r
        do if the user wants to avoid the loss of data
           then if the foreign key column (FK) is optional
                then exec SQL
                        update E1
                          set FK = null
                     end exec
                else call Delete_on_cascade(r1, E1)
           else call Delete_on_cascade(r1, E1);
  exec SQL  (* delete r from E *)
      delete
        from E
        where id = r.id
  end exec;
end;


exec SQL
   (* We have to delete all the rows except one of table E1 among those
      having the same value for R_a21. We have however first to 'remove on
      cascade' the rows of table E2 referencing the rows of E1 that will
      be deleted. *)

      declare c1 cursor for
        select R_a21, count(*)
          from E1
          group by R_a21
          having count(*) > 1
          order by R_a21 ASC;
      declare c2 cursor for
        select R_a21, a11
          from E1
          group by R_a21, a11
          order by R_a21 ASC, a11 ASC;
      open c1;
      open c2;
      fetch c2 into :a21_2, :a11;
      fetch c1 into :a21_1, :count;
end exec;
while SQLCODE = 0 (* the last item has not yet been treated *)
do begin
      while a21_1 <> a21_2
      do exec SQL
            fetch c2 into :a21_2, :a11
         end exec;
      exec SQL
            fetch c2 into :a21_2, :a11
      end exec;
      while (a21_1 = a21_2) and (SQLCODE = 0)
      do begin
```

```
            if (x = 0) and the user wants to avoid the loss of data
               wherever it is possible
            then exec SQL
                        update E1
                          set R_a21 = null
                          where a11 = :a11;
                 end exec
            else call Delete_on_cascade(row of current of c2, E1);
            exec SQL
              fetch c2 into :a21_2, :a11;
            end exec;
         end;
      exec SQL
            fetch c1 into :a21_1, :count
      end exec;
   end;
exec SQL
      close c1;
      close c2;
      (* we add the unique key feature to column R_a21 *)
      alter table E1
        add constraint unique (R_a21) constraint idE1_#;
end exec;
```

As each value in column R_a21 in relation E1 must be unique, we necessarily loose data when duplicate values appear in that column.

2.3.2.1.3. Program Extracts

Note:
The modifications suggested here below are not absolutely necessary. They may be seen as optimizations.

- As R_a21 is now a unique key, for some select queries, we do not need any cursor. In addition, certain loops (for example while-loops) should be replaced by simple if-then tests.

```
var ...
 :
exec SQL
  declare c cursor for
    select ...
      from E1
      where R_a21 = ...;
  open c;
  fetch c into ...;
end exec;
while SQLCODE = 0
do begin
    :
  exec SQL
       fetch c into ...
  end exec;
  end;
exec SQL
  close c
end exec;
 :
```

```
var ...

 :
exec SQL
   select ...
     into ...
     from E1
     where R_a21 = ...;
end exec;
if SQLCODE = 0
then ...
 :
```

- For the same reason , most of the functions (min, max, distinct, ...) can be dropped. For example:

```
select distinct ...
  from E1
  where R_a21 = ...
```

```
select ...
  from E1
  where R_a21 = ...
```

- Note that certain user interfaces and variables must also be adapted.

2.3.2.2. Augment_min_card

Precondition:
Given the restrictions of the relationship-types in the Kernel (see page 3-2), the only augmentations of the minimum cardinality of a role that we accept so far are:

- 0-1/0-1 → 1-1/0-1
- 0-1/0-N → 1-1/0-N

We want to augment to 1 the minimum cardinality of the 0-1 role of R played by entity-type E1.

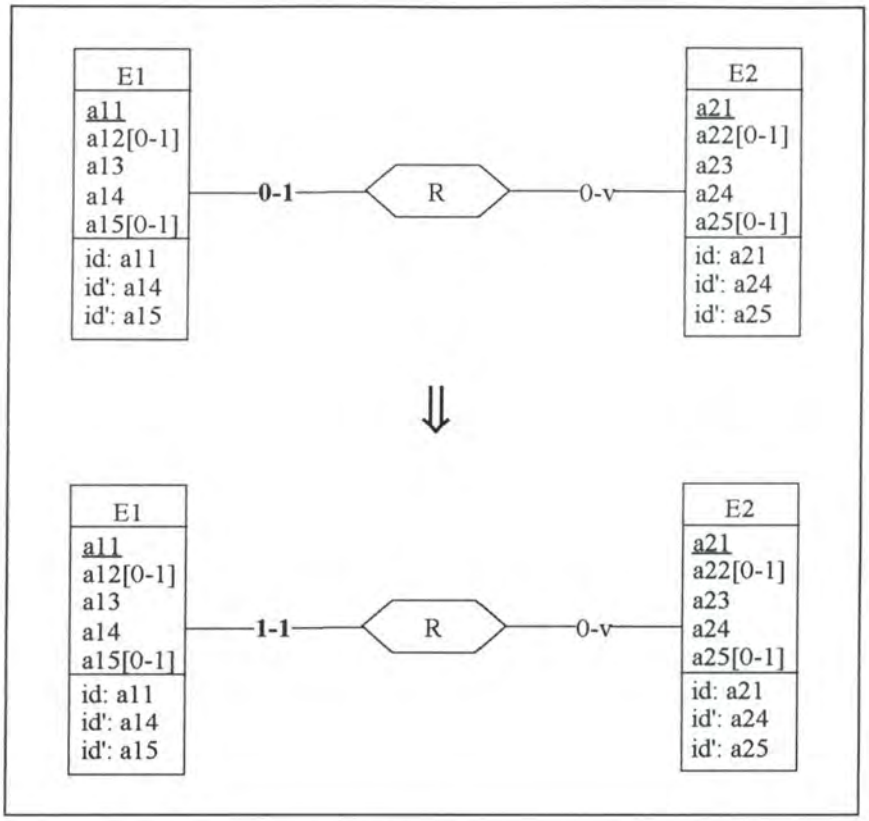


Figure A2 - 36 : Augmenting the minimum cardinality of a role to 1 on the conceptual level

2.3.2.2.1. Logical Schema

We either make the foreign key R_a21 in E1 mandatory or we replace the foreign key R_a11 in E2 by a mandatory foreign key R_a21 in E1.

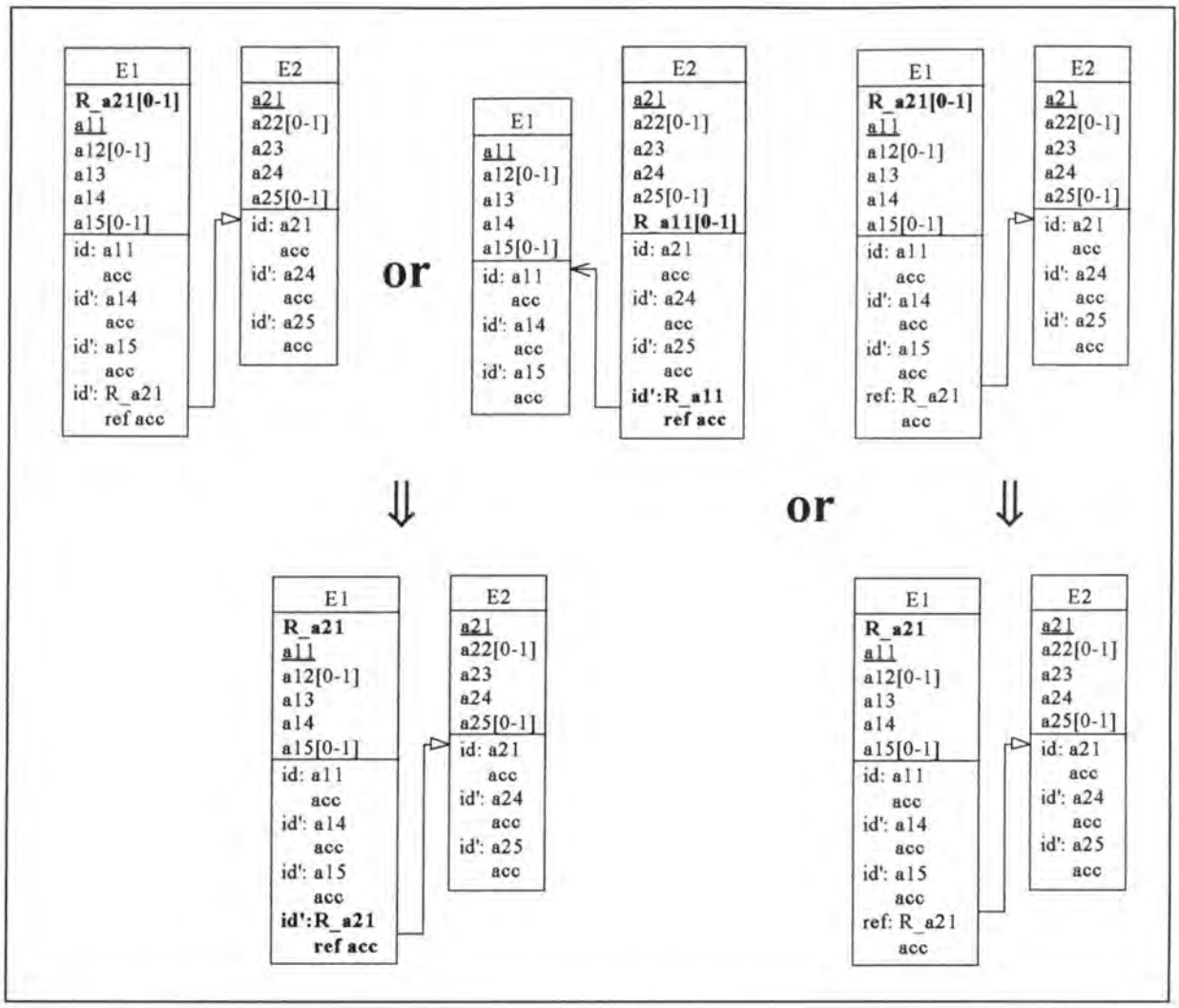


Figure A2 - 37 : Augmenting the minimum cardinality of a role to 1 on the logical level

2.3.2.2.2. SQL Description & Data

```
var a21: <type>;
    a11: <type>;

procedure Delete_on_cascade(r, E);
(* Before deleting a row r in table E, we must delete the rows r1
   'referencing r' or set to null in table E1 the foreign key column of the
   rows r1 'referencing r'. If the rows r1 are deleted, the problem must be
   treated recursively. *)

begin
  for each table E1
  do for each foreign key referencing table E
     do for each of the rows r1 having as foreign column value the value of
           the primary key column of row r
        do if the user wants to avoid the loss of data
           then if the foreign key column (FK) is optional
                then exec SQL
                       update E1
                         set FK = null
                     end exec
                else call Delete_on_cascade(r1, E1)
```

```
          else call Delete_on_cascade(r1, E1);
  exec SQL  (* delete r from E *)
      delete
        from E
        where id = r.id
  end exec;
end;


if R is implemented by a foreign key in E1
then begin
      (* we 'remove on cascade' all the rows which are not linked to E2 *)
        for each row r1 of E1 where R_a21 is null
        do call Delete_on_cascade(r1, E1);
        exec SQL
            alter table E1
              alter R_a21  not null  constraint E1_R_a21;
        end exec;
      end
else (* R is implemented by a foreign key in E2 *)
      begin
        exec SQL
            (* we create the new foreign key column *)
            alter table E1
              add R_a21  <type>  default <value>  not null
                                                  constraint E1_R_a21;
            (* we copy the data representing relationship-type R from table
               E2 into table E1 *)
            declare c cursor for
              select a21, R_a11
                from E2
                where R_a11 is not null;
            open c;
            fetch c into :a21, :a11;
        end exec;
        while SQLCODE = 0     (* the last item has not yet been treated *)
        do begin
              exec SQL
                  update E1
                    set R_a21 = :a21
                    where a11 = :a11;
                  fetch c into :a21, :a11;
              end exec;
            end;
        (* we 'remove on cascade' all the rows of E1 which are not linked
           to E2 *)
        for each of the rows r of table E1 where R_a21 = <value>
        do call Delete_on_cascade(r, E1);
        exec SQL
            close c;
            (* we add and remove the necessary constraints *)
            alter table E1
              add constraint unique (R_a21)  constraint idE1_#;
              add constraint foreign key (R_a21) references E2
                                                  constraint E2_#;
            alter table E2
              drop constraint idE2_#,      (* we remove the old unique key
                                              feature *)
              drop constraint E1_#,        (* we remove the old foreign key
                                              feature *)
              drop R_a11;
        end exec;
      end;
```

This modification involves loss of data, as we drop the rows from E1 which have a null or a default value for column R_a21.

2.3.2.2.3. Program Extracts

In some cases select queries referencing the foreign key representing relationship-type R must be deleted or modified. For example:

```
select ...
  from E2
  where R_a11 = ...
```

```
select ...
  from E2
  where a21 in (select R_a21
                from E1
                where a11 = ...)
```

It is often however not sufficient to modify or delete the select queries only. The application programs in which they appear should also be reviewed. Certain variables can be deleted and tests on the null value of column R_a21 can be dropped.

```
var a21: <type>;
    null_indicator: INTEGER;

  :
exec SQL
      select R_a21, ...
        into :a21:null_indicator, ...
        from E1
        where a11 = ...;
end exec;
if SQLCODE = 0     (* if such a row has been found *)
then if null_indicator = 0
     then...
```

```
var a21: <type>;

  :
exec SQL
      select R_a21, ...
        into :a21, ...
        from E1
        where a11 = ...;
end exec;
if SQLCODE = 0     (* if such a row has been found *)
then ...
```

2.4. MODIFICATIONS OF THE ATTRIBUTES

2.4.1. Modifications which Augment the Semantics

2.4.1.1. Add_optional_attribute

Let us suppose we want to add an optional attribute a16 to E1.

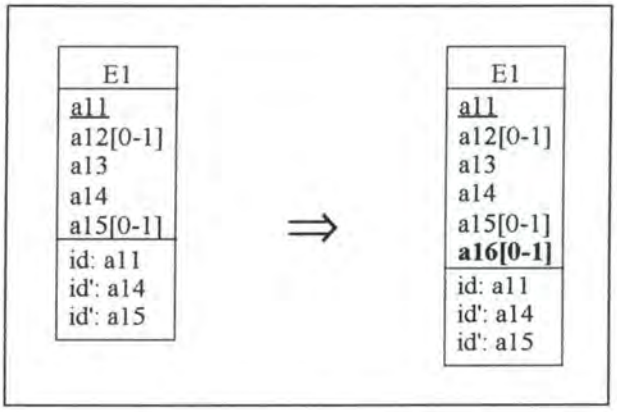


Figure A2 - 38 : Adding an optional attribute on the conceptual level

2.4.1.1.1. Logical Schema

We add an optional column a16 to the relation E1.

2.4.1.1.2. SQL Description & Data

```
alter table E1
  add a16  <type>;
```

Note that all the rows of E1 have a null value for column a16.

2.4.1.1.3. Program Extracts

For some select queries, we have either to add a variable corresponding to column a16 or to change the select clause.

```
var a11: <type>;
      :
    a15: <type>;

      :
   exec SQL
        select *
          into :a11, ..., :a15
          from E1
```

```
                                    where a11 = ...;
                                end exec
                                  :
```

```
var a11: <type>;                                  or        var a11: <type>;
      :                                                           :
    a15: <type>;                                              a15: <type>;
    a16: <type>;

  :                                                           :
exec SQL                                                    exec SQL
     select *                                                 select a11, ..., a15
       into :a11, ..., :a15, :a16                              into :a11, ..., :a15
       from E1                                                 from E1
       where a11 = ...;                                        where a11 = ...;
end exec                                                    end exec
  :                                                           :
```

A similar remark can be formulated for 'select * from E' queries which occur in cursor declarations. Moreover, we have to change the application programs, for instance, by assigning an output field for a16 in the user interfaces.

2.4.1.2. Add_mandatory_attribute

Let us suppose we want to add as well a mandatory attribute a17 to E1.

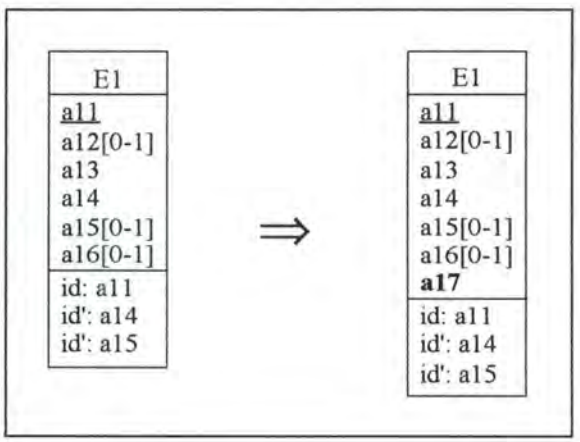


Figure A2 - 39 : Adding a mandatory attribute on the conceptual level

2.4.1.2.1. Logical Schema

We add a column a17 to the relation E1.

2.4.1.2.2. SQL Description & Data

```
alter table E1
  add a17  <type>  default <value>  not null  constraint E1_a17;
```

In order to keep all the data of E1, we place the default value <value> in column a17 for each row of the table E1.

2.4.1.2.3. Program Extracts

Similar remarks can be formulated as for the case add_optional_attribute (see page A2-50).

2.4.1.3. Make_attr_optional

Precondition:
The attribute that should be made optional must not be a primary key.

Let us suppose that we want to make a17 in E1 optional.

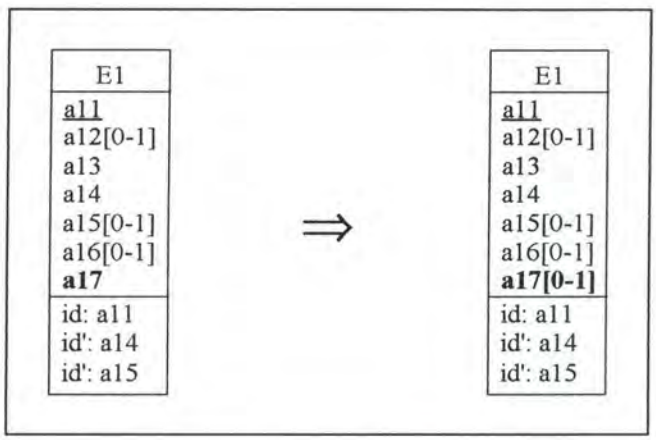


Figure A2 - 40 : Making an attribute optional on the conceptual level

2.4.1.3.1. Logical Schema

We make column a17 optional in relation E1.

2.4.1.3.2. SQL Description & Data:

```
alter table E1
  drop constraint E1_a17;
```

Note that the data will not be changed as we only make the column a17 optional.

2.4.1.3.3. Program Extracts

In certain program extracts, we have now to test whether the result of a select query is null or not.

```
var a17: <type>;

  :
exec SQL
      select a17
        into :a17
        from E1
        where a11 = ...
end exec;
if SQLCODE = 0
then ...
```

```
var a17: <type>;
    null_indicator: INTEGER;

  :
exec SQL
      select a17
        into :a17:null_indicator
        from E1
        where a11 = ...
end exec;
if SQLCODE = 0        (* if such a row has been found *)
then if null_indicator = 0
     then ...
```

2.4.1.4. Extend_domain_attribute

Precondition:
The attribute whose domain should be modified must not be an identifier. This is due to SQL-RDB which allows modifications only on columns, on which no constraints (primary, unique and foreign key) apply. In addition, the attribute cannot be of the type date.

Let us suppose we want to extend the domain of column a13 in entity-type E1 from type(x) to type(y) where y > x.

2.4.1.4.1. Logical Schema

We extend the domain of column a13 in table E1 from type(x) to type(y) where y > x.

2.4.1.4.2. SQL Description & Data

```
if a13 is mandatory
then exec SQL
           alter table E1
             drop constraint E1_a13,      (* we remove the mandatory
                                             feature from column a13 *)
             alter a13 type(y) not null constraint E1_a13;
     end exec
else (* a13 is optional *)
     exec SQL
           alter table E1
             alter a13 type(y);
     end exec;
```

No modifications are made on the data.

2.4.1.4.3. Program Extracts

In the application programs the variables, the procedure arguments and the user interface output fields referencing column a13 of table E1 must be adapted accordingly. For example:

```
var a13: TYPE[x];

  :
exec SQL
      select a13
        into :a13
        from E1
        where a11 = ...
end exec;
  :
```

```
var a13: TYPE[y];

  :
exec SQL
      select a13
        into :a13
        from E1
        where a11 = ...
end exec;
  :
```

2.4.1.5. Change_type_int_char

Precondition:
The attribute must not be an identifier (neither a primary nor a unique key). This is due to SQL-RDB which allows modifications only on columns, on which no constraints (primary, unique and foreign key) apply.

Let us suppose that we want to change the type integer of attribute a13 to char(11) in entity-type E1.

2.4.1.5.1. Logical Schema

We have to change the type integer of column a13 to char(11) in relation E1.

2.4.1.5.2. SQL Description & Data

```
var i: INTEGER;
    s: STRING[11];

exec SQL
      (* we create an intermediate column *)
      alter table E1
```

```
        add a integer;
      (* we copy the data of column a13 into that column *)
      update E1
        set a = a13;
end exec;
if a13 is mandatory
then exec SQL
           (* the old column a13 is replaced by a new one *)
           alter table E1
             drop constraint E1_a13,      (* we remove the mandatory
                                             feature from column a13 *)
             drop a13,
             add a13  char(11)  default '0'  not null constraint E1_a13;
           declare c cursor for
             select a
               from E1
             for update of a13 in E1;
      end exec
else (* a13 is optional *)
      exec SQL
           (* the old column a13 is replaced by a new one *)
           alter table E1
             drop a13,
             add a13  char(11);
           declare c cursor for
             select a
               from E1
               where a is not null
             for update of a13 in E1;
      end exec;
exec SQL
      open c;
      fetch c into :i;
end exec;
while SQLCODE = 0 (* the last item has not yet been treated *)
do begin
      (* the data is converted and copied into the new column a13 *)
      s := f_int_char(i);
      exec SQL
           update E1
             set a13 = :s
             where current of c;
           fetch c into :i;
      end exec;
   end;
exec SQL
      close c;
      (* the intermediate column is dropped *)
      alter table E1
        drop a;
end exec;
```

f_int_char:
This function converts an integer into a string.

No data is lost, but we have to note that the values of column a13 are converted.

2.4.1.5.3. Program Extracts

In the application programs the user interface output fields, the variables, the procedure arguments and sometimes the constants referencing column a13 of E1 must be adapted accordingly. For example:

```
var a13: INTEGER;

  :
exec SQL
      select a13
        into :a13
        from E1
        where a11 = ...
end exec;
  :
```

```
var a13: STRING[11];

  :
exec SQL
      select a13
        into :a13
        from E1
        where a11 = ...
end exec;
  :
```

2.4.1.6. Change_type_float_char

This modification is similar to the previous one (see page A2-54), except that we use function f_float_char instead of f_int_char. Function f_float_char converts a float into a string.

2.4.1.7. Change_type_date_char

This modification is similar to change_type_int_char (see page A2-54), except that we use function f_date_char instead of f_int_char. Function f_date_char converts a date into a string.

2.4.1.8. Change_type_date_int

This modification is similar to change_type_int_char (see page A2-54), except that we use function f_date_int instead of f_int_char. Function f_date_int converts a date into an integer.

2.4.1.9. Change_type_int_float

This modification is similar to change_type_int_char (see page A2-54), except that we use function f_int_float instead of f_int_char. Function f_int_float converts an integer into a float.

2.4.1.10. Change_type_date_float

This modification is similar to change_type_int_char (see page A2-54), except that we use function f_date_float instead of f_int_char. Function f_date_float converts a date into a float.

2.4.2. Modifications which Decrease the Semantics

2.4.2.1. Remove_optional_attribute

Let us suppose that we want to remove the attribute a16 from the entity-type E1.

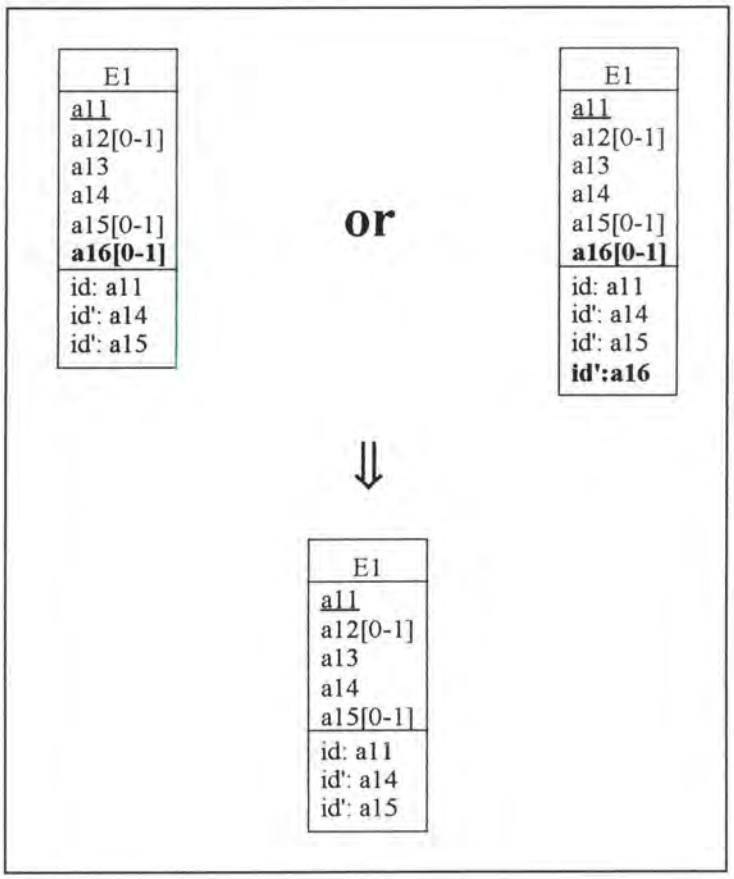


Figure A2 - 41 : Removing an optional attribute on the conceptual level

2.4.2.1.1. Logical Schema

We remove the column a16 from relation E1.

2.4.2.1.2. SQL Description & Data

```
if a16 is not a unique key
then exec SQL
           alter table E1
             drop a16;
     end exec
else (* a16 is a unique key *)
     exec SQL
           alter table E1
             drop constraint idE1_#,      (* we remove the unique key
                                             feature *)
             drop a16;
```

```
end exec;
```

All the data of column a16 will be lost.

2.4.2.1.3. Program Extracts

It is often not sufficient to delete or modify the select queries referencing a16. The application programs in which they appear must also be reviewed: certain variables may be dropped and certain user interfaces may be adapted.

2.4.2.2. Remove_mandatory_attribute

Precondition:
The attribute which should be removed must not be a primary key and must not be the last attribute of the entity-type.

Let us imagine we want to remove attribute a13 from entity-type E1.

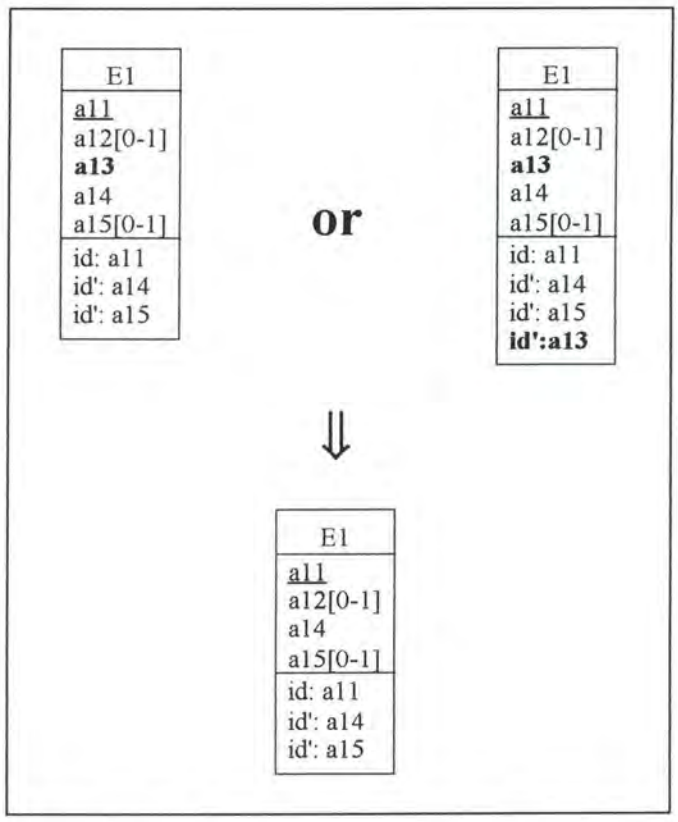


Figure A2 - 42 : Removing a mandatory attribute on the conceptual level

2.4.2.2.1. Logical Schema

We remove the column a13 from relation E1.

2.4.2.2.2. SQL Description & Data

```
if a13 is not a unique key
then exec SQL
            alter table E1
              drop constraint E1_a13,      (* we remove the mandatory
                                              feature from column a13 *)
              drop a13;
      end exec;
else (* a13 is a unique key *)
      exec SQL
            alter table E1
              drop constraint idE1_#,      (* we remove the unique key
                                              feature *)
              drop constraint E1_a13,      (* we remove the mandatory
                                              feature from column a13 *)
              drop a13;
```

All the data of column a13 will be lost:

2.4.2.2.3. Program Extracts

The remarks concerning the program extracts are similar to those of the modification remove_optional_attribute (see page A2-58).

2.4.2.3. Make_attr_mandatory

We have to distinguish whether the attribute which we want to make mandatory is a unique key or not. Let us suppose we want to make attribute a12 in entity-type E1 mandatory.

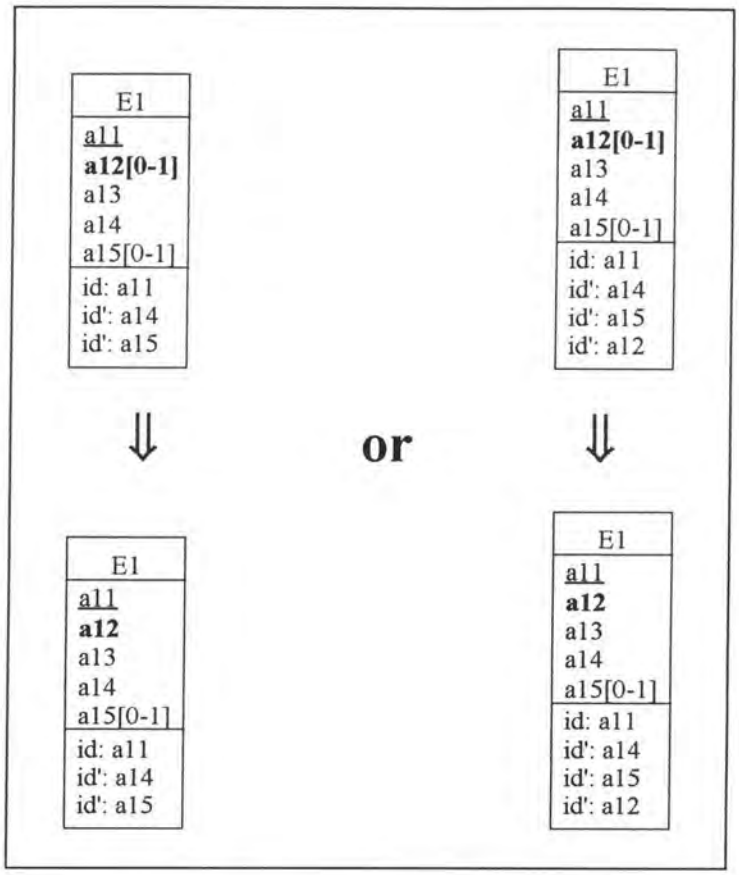


Figure A2 - 43 : Making an attribute mandatory on the conceptual level

2.4.2.3.1. Logical Schema

We make the column a12 in relation E1 mandatory.

2.4.2.3.2. SQL Description & Data

```
procedure Delete_on_cascade(r, E);

(* Before deleting a row r in table E, we must delete the rows r1
   'referencing r' or set to null in table E1 the foreign key column of the
   rows r1 'referencing r'. If the rows r1 are deleted, the problem must be
   treated recursively. *)

begin
  for each table E1
  do for each foreign key referencing table E
     do for each of the rows r1 having as foreign column value the value of
           the primary key column of row r
        do if the user wants to avoid the loss of data
           then if the foreign key column (FK) is optional
                then exec SQL
                        update E1
                          set FK = null
                     end exec
                else call Delete_on_cascade(r1, E1)
           else call Delete_on_cascade(r1, E1);
  exec SQL  (* delete r from E *)
```

```
      delete
        from E
        where id = r.id
  end exec;
end;


if a12 is not a unique key
then begin
       if the user wants to avoid the loss of data wherever it is possible
       then exec SQL
                  update E1
                    set a12 = <value>
                    where a12 is null
            end exec
       else for all the rows r of E1 having a null value for column a12
            do call Delete_on_cascade(r, E1);
       exec SQL
            alter table E1
              alter a12 not null constraint E1_a12;
       end exec;
     end
else begin
       (* we cannot use here a default value because of the unique key
          feature of column a12 *)
       for all the rows r of E1 having a null value for column a12
       do call Delete_on_cascade(r, E1);
       exec SQL
           alter table E1     (* we can only modify a column on which no
                                 constraints apply *)
             drop constraint idE1_#,     (* we remove the old unique key
                                            feature *)
             alter a12 not null constraint E1_a12,
             add constraint unique (a12) constraint idE1_#;
       end exec;
     end;
```

It depends on the choice of the user and on the uniqueness feature of the column a12 whether we loose data or not.

2.4.2.3.3. Program Extracts

Select queries testing the null value of the attribute that has to be made mandatory must be modified or deleted depending on the case.

```
select ...
  from E1
  where a12 is null
```

```
select ...
  from E1
  where a12 = <value>
```

or

(* The user did not want to loose data and a12 is not a unique key *)

(* The user accepted to loose data or a12 is a unique key *)

It is often not sufficient to change or delete the select queries only, we must also review the program extracts in which they appear. For example: in certain cases we do not need the null

indicator anymore and certain tests, checking the null value of column a12, must either be changed or dropped.

```
var a12: <type>;
      :
    null_indicator: INTEGER;

exec SQL
      select a12
        into :a12:null_indicator, ...
        from E1
        where a11 = ...
end exec;
if SQLCODE = 0     (* if such a row has been found *)
then if null_indicator = 0
     then ...
```

```
var a12: <type>;
      :

exec SQL
      select a12
        into :a12, ...
        from E1
        where a11 = ...
end exec;
if SQLCODE = 0     (* if such a row has been found *)
then ...
```

2.4.2.4. Restrict_domain_attribute

Precondition:
The attribute whose domain should be modified must not be an identifier. This is due to SQL-RDB which allows modifications only on columns, on which no constraints (primary, unique and foreign key) apply. In addition, the attribute cannot be of the type date.

Let us suppose we want to restrict the domain of attribute a13 in entity-type E1 from type(x) to type(y) where y <x.

2.4.2.4.1. Logical Schema

We restrict the domain of column a13 in table E1 from type(x) to type(y) where y < x.

2.4.2.4.2. SQL Description & Data

```
if a13 is mandatory
then exec SQL
           alter table E1
             drop constraint E1_a13,      (* we remove the mandatory
                                             feature from column a13 *)
             alter a13 type(y) not null constraint E1_a13;
     end exec
else (* a13 is optional *)
     exec SQL
           alter table E1
             alter a13 type(y);
```

```
      end exec;
```

SQL-RDB truncates values already stored in the database that exceed the capacity of the new data type, but only when it retieves those values. (The values are not truncated in the database, however, until they are updated. If you only retrieve data, therefore, you can change the data type back to the original, and SQL again retrieves the entire original value.)[RDB91, page 7-48]

2.4.2.4.3. Program Extracts

In the application programs the variables, the procedure arguments and the user interface output fields referencing column a13 of E1 have to be adapted accordingly. For example:

```
var a13: TYPE[x];

  :
exec SQL
      select a13
        into :a13
        from E1
        where a11 = ...
end exec;
  :
```

```
var a13: TYPE[y];

  :
exec SQL
      select a13
        into :a13
        from E1
        where a11 = ...
end exec;
  :
```

2.4.2.5. Change_type_char_int

This modification is similar to change_type_int_char (see page A2-54), except that we use function f_char_int instead of f_int_char. Function f_char_int converts a string into an integer. Depending on the implementation of function f_char_int, we could loose data.

2.4.2.6. Change_type_float_int

This modification is similar to change_type_int_char (see page A2-54), except that we use function f_float_int instead of f_int_char. Function f_float_int converts a float into an integer. Depending on the implementation of function f_float_int, we could loose data.

2.4.2.7. Change_type_char_float

This modification is similar to change_type_int_char (see page A2-54), except that we use function f_char_float instead of f_int_char. Function f_char_float converts a string into a float. Depending on the implementation of function f_char_float, we could loose data.

2.4.2.8. Change_type_char_date

This modification is similar to change_type_int_char (see page A2-54), except that we use function f_char_date instead of f_int_char. Function f_char_date converts a string into a date. Depending on the implementation of function f_char_date, we could loose data.

2.4.2.9. Change_type_int_date

This modification is similar to change_type_int_char (see page A2-54), except that we use function f_int_date instead of f_int_char. Function f_int_date converts an integer into a date. Depending on the implementation of function f_int_date, we could loose data.

2.4.2.10. Change_type_float_date

This modification is similar to change_type_int_char (see page A2-54), except that we use function f_float_date instead of f_int_char. Function f_float_date converts a float into a date. Depending on the implementation of function f_float_date, we could loose data.

2.4.3. Modifications which Preserve the Semantics

2.4.3.1. Rename_optional_attribute

Let us rename a12 into a16 in entity-type E1. We have to distinguish whether the optional attribute is a unique key or not.

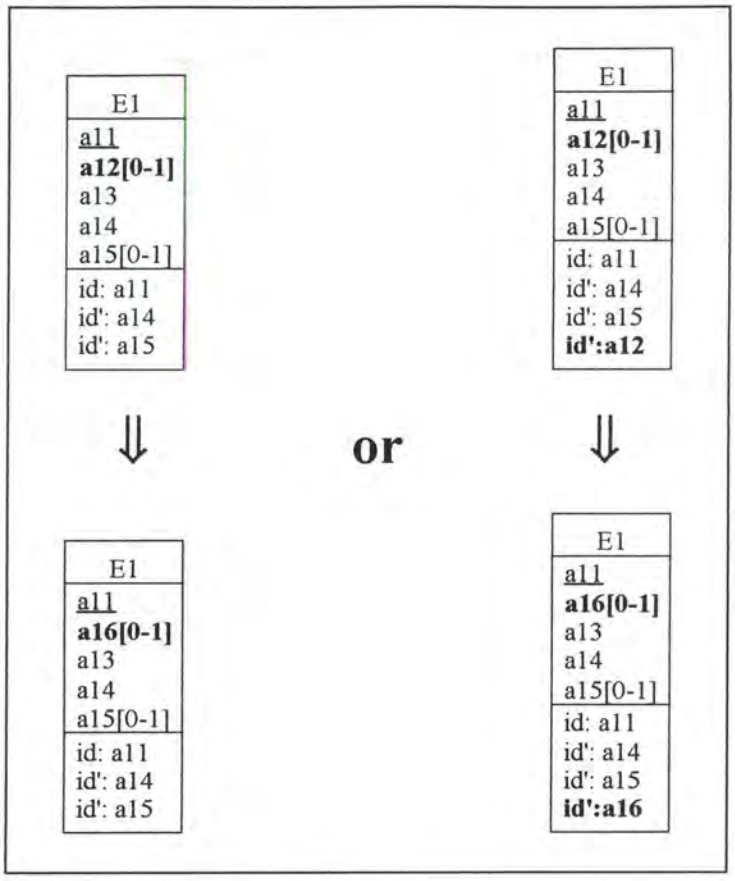


Figure A2 - 44 : Renaming an optional attribute on the conceptual level

2.4.3.1.1. Logical Schema

We rename a12 into a16 in relation E1.

2.4.3.1.2. SQL Description & Data

```
exec SQL
      alter table E1
        add a16  <type>;
      update E1
        set a16 = a12;
end exec;
if a12 is not a unique key
then exec SQL
           alter table E1
             drop a12;
      end exec
else (* a12 is a unique key *)
      exec SQL
           alter table E1
             drop constraint idE1_#,      (* we remove the unique key
                                             feature *)
             drop a12,
             add constraint unique(a16) constraint idE1_#;
      end exec;
```

Note:
In order to avoid copying a whole column, we can create a view. But as the view name must be unique among all view and table names in the schema, we would have to change all the select queries referencing that table. We thus prefer the first approach though it is rather slowly to be executed.

No data is lost as we only copy the data from one column into another.

2.4.3.1.3. Program Extracts

- In fact, in every query referencing a12, it must be replaced by a16.

```
select ...
  from E1
  where a12 ...
```

```
select ...
  from E1
  where a16 ...
```

- Sometimes it might be good to rename also certain labels of the user interface output fields and certain variables accordingly.

```
var a12: <type>;

exec SQL
    select a12
      into :a12
      from E1
      where a11 = ...
end exec
```

```
var a16: <type>;

exec SQL
    select a16
      into :a16
      from E1
      where a11 = ...
end exec
```

2.4.3.2. Rename_mandatory_attribute

Precondition:
In order to avoid having also to rename the foreign keys, the attribute which should be renamed must not be a primary key.

Let us rename a13 into a17 in entity-type E1. We distinguish whether the attribute is a unique key or not.

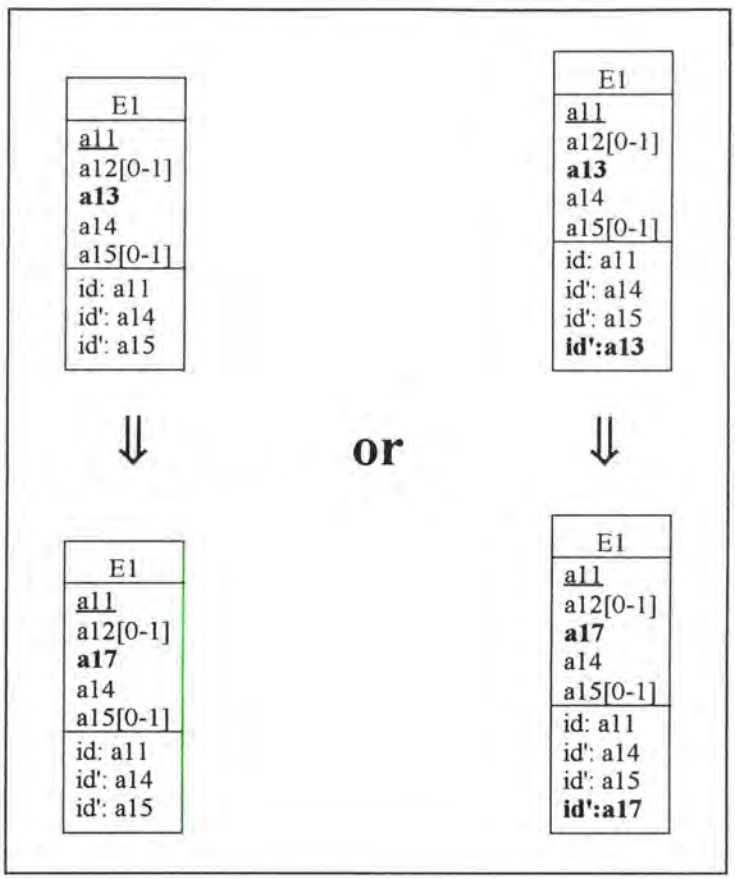


Figure A2 - 45 : Renaming a mandatory attribute on the conceptual level

2.4.3.2.1. Logical Schema

We rename the column a13 into a17 in relation E1.

2.4.3.2.2. SQL Description & Data

```
exec SQL
      alter table E1
        add a17  <type> default <value> not null constraint E1_a17;
      update E1
        set a17 = a13;
end exec;
if a13 is not a unique key
then exec SQL
           alter table E1
             drop constraint E1_a13,      (* we remove the mandatory
                                             feature from column a13 *)
             drop a13;
      end exec
else (* a13 is a unique key *)
      exec SQL
           alter table E1
             drop constraint idE1_#,      (* we remove the old unique key
                                             feature *)
```

```
            drop constraint E1_a13,      (* we remove the mandatory
                                            feature from column a13 *)
            drop a13,
            add constraint unique(a17) constraint idE1_#;
    end exec;
```

Note:
In order to avoid copying a whole column, we can create a view. But as the view name must be unique among all view and table names in the schema, we would have to change all the select queries referencing that table. We thus prefer the first approach though it is rather slowly to be executed.

No data is lost as we only copy the data from one column into another.

2.4.3.2.3. Program Extracts:

Similar remarks can be formulated as for the case rename_optional_attribute (see page A2-66).

2.5. MODIFICATIONS OF THE IDENTIFIER

2.5.1. Modifications which Augment the Semantics

2.5.1.1. Remove_unique_feature

Let us suppose we want to remove the uniqueness constraint from a13 in entity-type E1.

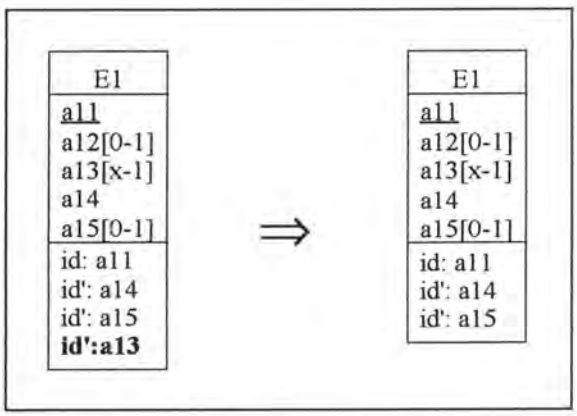


Figure A2 - 46 : Removing a unique key feature on the conceptual level

2.5.1.1.1. Logical Schema

We remove the uniqueness constraint from a13 in relation E1.

2.5.1.1.2. SQL Description & Data

```
alter table E1
  drop constraint idE1_#;
```

No changes are made on the data.

2.5.1.1.3. Program Extracts

As a13 has lost its uniqueness feature, in some cases, we have to define a cursor.

```
var a11: <type>

  :
exec SQL
      select a11
        into :a11
        from E1
        where a13 = ...;
end exec;
if SQLCODE = 0
```

```
then ...
  :
```

```
var a11: <type>;

  :
exec SQL
      declare c cursor for
        select a11
          from E1
          where a13 = ...;
      open c;
      fetch c into :a11;
end exec;
while SQLCODE = 0 (* the last item has not yet been treated *)
do begin
        :
      exec SQL
            fetch c into :a11
      end exec;
   end;
exec SQL
      close c;
end exec;
  :
```

As we can see in the previous program extracts, simple test conditions must be transformed into loops. Note that certain user interfaces must also be adapted - for example by inserting list boxes.

2.5.2. Modifications which Decrease the Semantics

2.5.2.1. Add_unique_feature

Let us suppose we want to make a13 a unique key of E1.

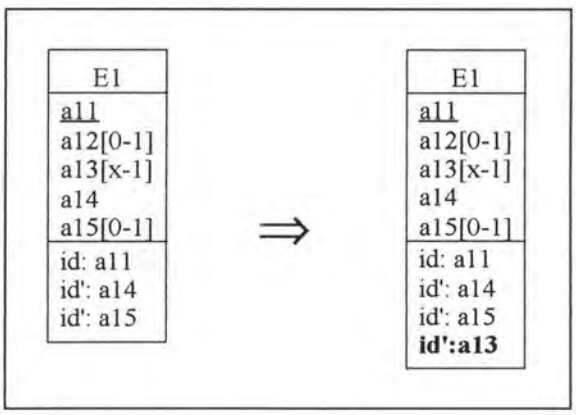


Figure A2 - 47 : Adding a unique key feature on the conceptual level

2.5.2.1.1. Logical Schema

We have to add the unique key feature to column a13 in relation E1.

2.5.2.1.2. SQL Description & Data

```
var a13_1: <type>;
    a13_2: <type>;
    a11: <type>;
    count: INTEGER;

procedure Delete_on_cascade(r, E);

(* Before deleting a row r in table E, we must delete the rows r1
   'referencing r' or set to null in table E1 the foreign key column of the
   rows r1 'referencing r'. If the rows r1 are deleted, the problem must be
   treated recursively. *)

begin
  for each table E1
  do for each foreign key referencing table E
     do for each of the rows r1 having as foreign column value the value of
           the primary key column of row r
        do if the user wants to avoid the loss of data
           then if the foreign key column (FK) is optional
                then exec SQL
                        update E1
                          set FK = null
                     end exec
                else call Delete_on_cascade(r1, E1)
           else call Delete_on_cascade(r1, E1);
  exec SQL  (* delete r from E *)
      delete
        from E
        where id = r.id
  end exec;
end;


(* We have to delete or to set to null all the rows except one of table E1
   among those having the same value for a13. In case we want to delete
   those rows, we must first 'remove on cascade' the rows of tables
   referencing these rows. *)

if x = 1 (* a13 is mandatory *)
then exec SQL
      declare c1 cursor for
        select a13, count(*)
          from E1
          group by a13
          having count(*) > 1
          order by a13 ASC;
      end exec
else (* a13 is optional *)
      exec SQL
      declare c1 cursor for
        select a13, count(*)
          from E1
          where a13 is not null
          group by a13
          having count(*) > 1
          order by a13 ASC;
      end exec;
exec SQL
      declare c2 cursor for
        select a13, a11
          from E1
```

```
          group by a13, a11
          order by a13 ASC, a11 ASC;
       open c1;
       open c2;
       fetch c2 into :a13_2, :a11;
       fetch c1 into : a13_1, :count;
end exec;
while SQLCODE = 0 (* the last item has not yet been treated *)
do begin
       while a13_1 <> a13_2
       do exec SQL
             fetch c2 into :a13_2, :a11;
          end exec;
       exec SQL
           fetch c2 into :a13_2, :a11;
       end exec;
       while (a13_1 = a13_2) and (SQLCODE = 0)
       do begin
             if (x = 0) and the user wants to avoid the loss  of data
                wherever it is possible
             then exec SQL
                         update E1
                           set a13 = null
                           where a11 = :a11;
                  end exec
             else call Delete_on_cascade(row of current of c2, E1);
             exec SQL
                   fetch c2 into :a13_2, :a11;
             end exec;
          end;
       exec SQL
             fetch c1 into :a13_1, :count;
       end exec;
   end;
exec SQL
       close c1;
       close c2;
       (* we add the unique key feature to column a13 *)
       alter table E1
         add constraint unique (a13) constraint idE1_#;
end exec
```

We loose either data in column a13 only (if we set certain values of column a13 to null), or even whole rows where duplicate values for column a13 in relation E1 appear. In addition, if there are tables referencing the deleted rows in table E1, there could be even more loss of data.

2.5.2.1.3. Program Extracts

- As a13 becomes a unique key, in some cases, we do not need cursors anymore. As we can see in the following program extracts, loops may be transformed into simple test conditions.

```
var a11: <type>;

   :
exec SQL
   declare c cursor for
     select a11
       from E1
       where a13 = ...;
   open c;
   fetch c into :a11;
end exec;
while SQLCODE = 0 (* the last item has not yet been treated *)
do begin
```

```
    :
  exec SQL
       fetch c into :a11
  end exec;
   end;
exec SQL
  close c;
end exec;
  :
```

```
var a11: <type>

  :
exec SQL
  select a11
    into :a11
    from E1
    where a13 = ...;
end exec;
if SQLCODE = 0
then ...
  :
```

- For the same reason, most of the functions (min, max, distinct, ...) can be dropped.

```
select distinct...
  from E1
  where a13 = ...
```

```
select ...
  from E1
  where a13 = ...
```

- Note that certain user interfaces should also be adapted - for example by replacing list boxes with simple display fields.

2.5.3. Modifications which Preserve the Semantics

2.5.3.1. Switch_PK_unique

We want to transform the existing primary key into a unique key in entity-type E1 and vice versa. The user has the choice whether to specify a unique key or not. If he does not specify any unique key, then a technical identifier is created as primary key.

Precondition:
If a unique key is specified then it must not be optional as SQL-RDB does not allow optional attributes as primary key.

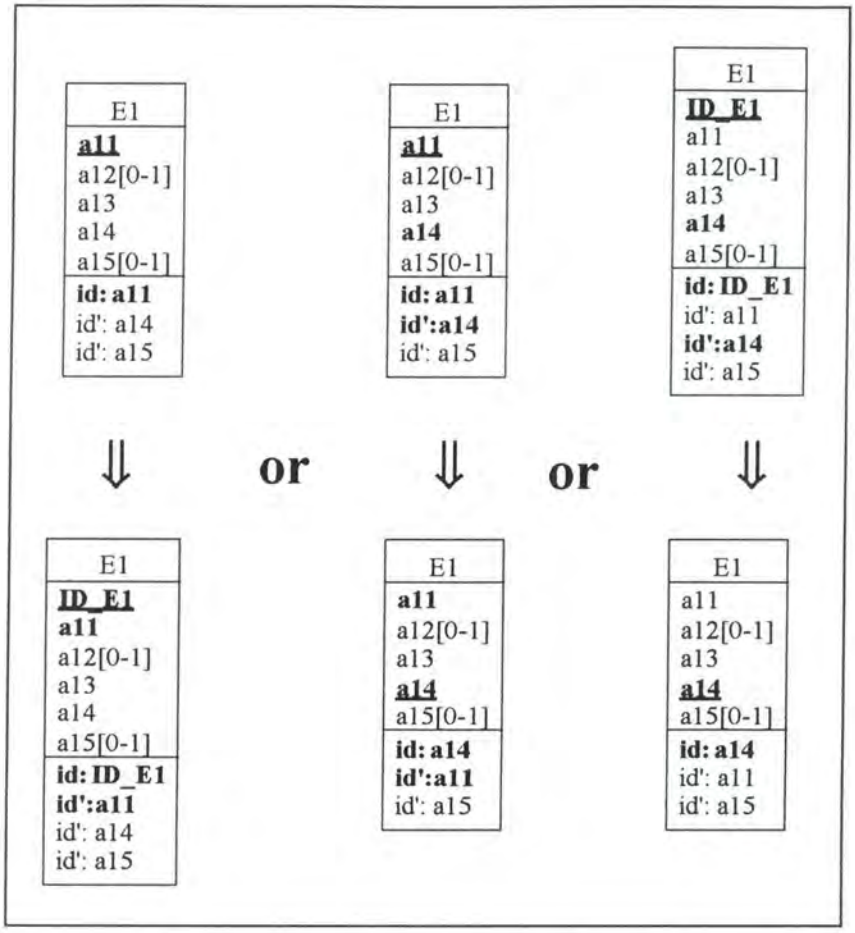


Figure A2 - 48 : Switching the primary key and the unique key on the conceptual level

2.5.3.1.1. Logical Schema

We transform the existing primary key into a unique key in relation E1, drop it if it was a technical one, create a technical primary key if no unique key was specified and replace the foreign keys referencing relation E1 accordingly.

2.5.3.2. SQL Description & Data

```
var i: INTEGER;
    idADD: INTEGER;

procedure Switch(E1, old_prim, new_prim)

(* This procedure transforms the existing primary key old_prim into a
   unique key in table E1 and the unique key new_prim into the new primary
   key of table E1. *)
```

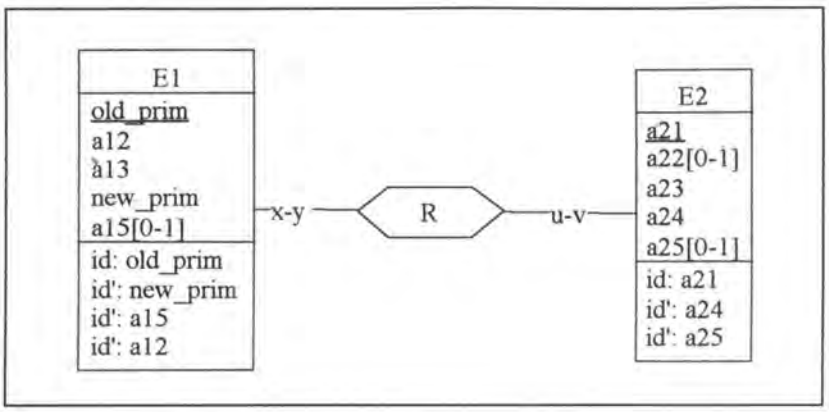


Figure A2 - 49 : General situation used in procedure Switch

```
var old_prim: <type>;
    new_prim: <type>;

begin
      for each foreign key in table E referencing table E1
          and representing relationship-type R
      do begin
            (* we create the new foreign key column, we remove the old
               foreign key constraint and we copy the data representing
               relationship-typ R *)
            if u = 0
            then exec SQL
                        alter table E
                          add R_new_prim <type>,
                          drop constraint E1_#;
                        declare c cursor for
                          select R_old_prim
                            from E
                            where R_old_prim is not null
                          for update of R_new_prim;
                  end exec
            else (* u = 1 *)
                  exec SQL
                        alter table E
                          add R_new_prim <type> default <value>  not null
                              constraint  E_R_new_prim,
                          drop constraint E1_#;
                        declare c cursor for
                          select R_old_prim
                            from E
                          for update of R_new_prim;
                  end exec;
            exec SQL
                  open c;
                  fetch c into :old_prim;
            end exec;
            while SQLCODE = 0 (* the last item has not yet been treated *)
            do exec SQL
                  select new_prim
                    into :new_prim
                    from E1
                    where old_prim = :old_prim;
                  update E
                    set R_new_prim = :new_prim
                    where current of c;
                  fetch c into :old_prim;
               end exec;
            exec SQL
                  close c
```

```
            end exec;
         end;
      if old_prim in E1 is a technical identifier
      then exec SQL
                (* we drop the old primary key with its constraints and
                   we add the primary key feature to column new_prim *)
                alter table E1
                  drop constraint idE1_#,
                                  (* primary key feature of old_prim *)
                  drop constraint E1_ID_E1,
                  drop ID_E1,
                  drop constraint idE1_#,
                                  (* uniqueness feature of new_prim *)
                  add constraint primary key (new_prim)
                                                  constraint idE1_#;
            end exec
      else (* old_prim in E1 is not a technical identifier *)
            exec SQL
                 (* We switch the identifying features between new_prim
                    and old_prim *)
                alter table E1
                  drop constraint idE1_#,
                                  (* primary key feature of old_prim *)
                  add constraint unique (old_prim) constraint idE1_#,
                  drop constraint idE1_#,
                                  (* uniqueness feature of new_prim *)
                  add constraint primary key (new_prim)
                                                  constraint idE1_#;
            end exec;
      for each foreign key in table E referencing table E1
          and representing relationship-type R
      do begin
            if u = 1 (* the old foreign key column R_old_prim was
                        mandatory *)
            then exec SQL
                        alter table E
                          drop constraint E_R_old_prim;
                 end exec;
            if y = 1 (* the old foreign key column R_old_prim was a unique
                        key *)
            then exec SQL
                        alter table E
                          drop constraint idE_#
                          add constraint unique (R_new_prim)
                                                  constraint idE_#;
                 end exec;
            exec SQL
                 (* we add the new foreign key constraint and remove the
                    old foreign key column *)
                 alter table E
                   add constraint foreign key (R_new_prim) references E1
                                                  constraint E1_#,
                   drop R_old_prim;
            end exec;
         end;
end; (* end of procedure *)


(* the program allows us to call the procedure 'Switch' with the correct
   arguments *)
if no unique key is specified
then begin
        exec SQL
          (* we create a technical identifier *)
          alter table E1
             add ID_E1 smallint  default 0 not null  constraint E1_ID_E1;
          (* we assign identifying values to that column *)
          declare c cursor for
```

```
            select ID_E1
              from E1
            for update of ID_E1 in E1;
          open c;
          fetch c;
        end exec;
        i:= 1;
        while SQLCODE = 0      (* the last item has not yet been treated *)
        do begin
              exec SQL
                  update E1
                    set ID_E1 = :i
                    where current of c;
                  fetch c;
              end exec;
              i:= i+1;
            end;
        exec SQL
            close c;
            (* we add the unique key feature to ID_E1 *)
            alter table E1
              add constraint unique (ID_E1) constraint idE1_#,
        end exec;
(* we operate the real switch *)
        call Switch(E1, a11, ID_E1);
      end
else (* a unique key is specified *)
      if the primary key of E1 is not a technical one
      then call Switch(E1, a11, a14)
      else call Switch(E1, ID_E1, a14);
```

No data is lost as we do not consider the information included in the technical identifier column ID_E1 as semantical data.

2.5.3.3. Program Extracts

Let us suppose we have switched primary key a11 with unique key a14.

Every select query which uses a foreign key referencing table E1 must be modified: we have to replace the foreign key.

-
```
select ...
  from E
  where R_a11 = c
```

⇓

```
select ...
  from E
  where R_a14 = d
```

-
```
select ...
  from E1
  where a11 in( select R_a11
                  from E
                  where ... )
```

⇓

```
select ...
  from E1
  where a14 in( select R_a14
                  from E
                  where ... )
```

A concrete example can be found in the modification switch_PK_unique in appendix 1 (see page A1-99). As we already said, it is not sufficient to change only the select queries. We must also review the application programs (for an example see page A1-103).